Migration of Software Components to Microservices: Matching and Synthesis

Andreas Christoforou, Lambros Odysseos and Andreas S. Andreou
Department of Electrical Engineering / Computer Engineering and Informatics, Cyprus University of Technology, 31 Archbishop Kyprianos Street, Limassol, Cyprus

Keywords: Software Engineering, Component Decomposition, Microservices, Ontology, Migration.

Abstract: Nowadays more and more software companies, as well as individual software developers, adopt the microservice architecture for their software solutions. Although many software systems are being designed and developed from scratch, a significant number of existing monolithic solutions tend to be transformed to this new architectural style. What is less common, though, is how to migrate component-based software systems to systems composed of microservices and enjoy the benefits of ease of changes, rapid deployment and versatile architecture. This paper proposes a novel and integrated process for the decomposition of existing software components with the aim being to fully or partially replace their functional parts with by a number of suitable and available microservices. The proposed process is built on semi-formal profiling and utilizes ontologies to match between properties of the decomposed functions of the component and those offered by microservices residing in a repository. Matching concludes with recommended solutions yielded by multi-objective optimization which considers also possible dependencies between the functional parts.

1 INTRODUCTION

Despite the differences in their approach and the time lag in their introduction to the software engineering community, Component-based Software Engineering (CBSE) (Cai et al., 2000), or, alternatively Component-based Development (CBD), and Microservices Architecture (MSA) (Dragoni et al., 2017) share the same inceptions, motivation and focus towards reuse of software artefacts. Both approaches aim at reducing complexity of the software development process, facilitate easy maintenance and support the operations for IT support. It may be argued that Service-Oriented Architecture (SOA) (Rosen et al., 2008), as the most recent emerging distributed development architecture, constitutes the common denominator between these two paradigms as it originated from component-based architecture and evolved to microservices architecture.

Following the new software engineering trends, Microservices architecture is tightly connected to the DevOps approach (Kleiner, 2009), which inherits its basic principles from agile methodologies and describes best practices to support the software development and operation processes. One may also argue that Microservices architecture actually supports the DevOps automation process and affects software engineering in a positive manner. To be more specific, it affects Software Engineering by introducing a different development approach. As regards how the DevOps process is automated, the latter relies primarily on the fact that the adoption of the Microservices architecture comprises a number of critical tasks than may be automated apart from the rest automated tasks like communication, coordination, monitoring, problem solving and deployment.

In recent years, Microservices architecture is gaining popularity in software development and the research community has turned its attention to related challenges (Esposito et al., 2016) such as the decomposition of a monolithic system into a set of independent services followed by synthesis of selected microservices to substitute their functionality. Microservice synthesis relies on locating and combining small functional service components which their characteristics match those of the decomposed system and put them in a proper order so as to meet the characteristics and the requirements of the initial monolithic system.

While literature includes a number of research works for decomposition approaches and migration from monolithic to microservice architecture, to the best of our knowledge, no work has been yet

Christoforou, A., Odysseos, L. and Andreou, A.
Migration of Software Components to Microservices: Matching and Synthesis.
DOI: 10.5220/0007732101340146
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 134-146
ISBN: 978-989-758-375-9

published that deals with software component decomposition and replacement of its functional parts with microservices. This paper aims to introduce an automatic process that identifies and recommends the full or partial replacement of a software component by a number of available microservices that support specific business operations. The proposed process adopts the basic principles proposed in a previous work of the authors (Andreou and Papatheocharous, 2015) related to a layered component-based software development architecture which was adapted and refined to accommodate the differences and peculiarities of the Microservices environment. The framework in (Andreou and Papatheocharous, 2015) supports the process of matching available components against a set of specifications expressed in a formalised syntax and utilizing ontologies. Apart from modifications to this framework, the present paper adds new tasks that extend and improve its recommendation layer.

The following research questions motivate and guide this research work:

RQ1: How should a Component Based Reusability Framework be modified to work equally well with Microservices?

RQ2: How can a component be decomposed into functional parts, independent or not?

RQ3: How can the assessment of the suitability of microservices to substitute component function be performed, and how should it handle cases where the decomposed functions present dependencies?

The remainder of the paper is organized as follows: A brief literature overview is performed in section 2, while section 3 describes our approach in detail. Section 4 presents the experimental process applied for evaluation and discusses the results produced. Finally, section 5 concludes the paper and outlines future research steps.

2 LITERATURE OVERVIEW

To the best of our knowledge this is the first attempt to propose a structured process targeting the decomposition of well described software components and replace the identified functional parts with microservices to the greatest possible degree. However, a brief literature overview has been carried out over the two topics that are strongly related to our research work, the software decomposition and the services synthesis.

Several different approaches have been proposed to deal with the decomposition of monolithic systems or services. Baresi, Garriga and De Renzis in (Baresi et al., 2017) propose a clustering-like approach to support the identification of microservices and the specifications of the extracted artefacts during either the design phase of a new system, or while the re-architecting of an existing system. Service Cutter (Gysel et al., 2016) is a tool framework that is based on a structured repeatable approach to decompose a monolith into microservices. A stepwise technique to identify microservices on monolithic systems is proposed in (Levcovitz et al., 2016) in which the authors deliver an approach based on a dependency graph among three distinct parts of an application, client, server and database. Balalaie, Heydarnoori and Jamshidi in (Barba, 2005) describe their experiences of an ongoing project on migrating an on-premise application to microservice architecture. Their approach is based on architectural refactoring, considering the characteristics of microservice architecture.

The vast majority of the literature which deals with services composition is concerned with web services. The work in (Moghaddam and Davis, 2015) presents a review of existing proposals for services selection by quoting the advantages and disadvantages of each approach. A systematical review of recent research on QoS-aware web service composition using computational intelligence techniques is presented in (Jatoth et al., 2015). A classification was developed for various research approaches along with the analysis of the different algorithms, mechanisms and techniques identified. An analysis and comparison of the latest representative approaches in the area of automated web service composition is the main contribution of the work in (Zeginis and Plexousakis, 2010). The existing research approaches were grouped into four distinct categories, workflow-based, model-based, mathematics-based and AI planning.

It is evident that the current literature on software services synthesis is limited, and especially in the case where this synthesis targets the migration from component-based development to microservices is rare if not non-existent. This is the gap the current paper aspires to fill.

3 AUTOMATIC SPECIFICATION AND MATCHING OF MICROSERVICES

3.1 Specification and Matching Framework

As previously mentioned, the present paper aims to

introduce an automatic process that identifies and recommends the full or partial replacement of a software component by a number of available microservices. The proposed process adopts basic principles proposed by the authors in a previous work (Andreou and Papatheocharous, 2015) More specifically, the utilization of the description layer in that work leverages the decomposition of a software component into distinct operations and respectively profiles all candidate microservices that may substitute these operations and simultaneously adhere to the same constraints (e.g. performance). Additionally, the translation of software components and microservices textual profiles into ontologies assists the automatic matching process towards the integrated replacement.

The proposed process follows the same 5-layers architecture as in our previous work (Andreou and Papatheocharous, 2015): (i) The Description layer provides a profile structure which includes all relevant information that describes the component(s) under decomposition and the available microservice(s). A developer/vendor of a component or microservice, defines a set of properties (functional and non-functional) that describe the specific artefact: (a) the component description which will serve as the basis for its decomposition and the properties characterizing each decomposing part, and, (b) the properties of a microservice that describe what it has to offer in terms of functionality, performance, availability, reliability, robustness etc. that one may look for when attempting to locate suitable microservices for integration and substitution of the component parts. (ii) The Location layer essentially provides general-purpose actions, like searching, locating and retrieving the microservice(s) of interest that match the profile of the component's decomposed parts. (iii) The Analysis layer evaluates the level of suitability of the candidate microservice(s) and provides matching results that will guide the selection of microservices for integration. (iv) The Recommendation layer uses the information provided by the previous layers and produces suggestions as to which of the candidate microservice(s) may be best integrated and why, based on an assessment made to ensure that certain requirements at the microservice level are preserved also at the integrated level. (vi) Finally, the Build level essentially comprises a set of integration and customization tools for combining component(s) to build larger systems. The present paper focuses on the first four layers and describes a novel way for automatic matching between desired and available microservice(s) based on the directions provided in (Andreou and Papatheocharous, 2015) and extending or revising them where appropriate. The interested reader may refer to that work for more details on the layered component architecture, whilst every effort has been made to make the current paper self-explanatory.

Components and microservices are first expressed in a semi-structured form of natural language which is then transformed into an ontology. This ontology standardises the description of the properties of the two software artefacts and will constitute the cornerstone of the specifications that will be used to match decomposed parts of components with the available microservice(s), the latter being stored in a repository. Thus, the problem of finding suitable microservice(s) to replace component functionality is reduced to matching (aligning) ontologies. This process is executed by automatically parsing the profile(s) of the two software artefacts (for simplicity we assume one component and N microservices) and their translation into instance values of two dedicated ontologies, one for each artefact, which are built so as to reflect the most critical properties suggested in literature for that artefact. At the same time, as the ontology of the component is being built, certain parts are marked so that a second step may then be executed which isolates these parts, as these are recognised to be directly comparable to parts of the microservices ontology. Hence, the latter step transforms them into a meta-ontology (subset of the component's initial ontology) describing the so-called 'required' or possible functions to be executed by available microservices. Then the matching of properties between the required and offered microservice(s) takes place automatically at the level of ontology items and a suitability ratio is calculated that suggests which microservice(s) to consider for possible integration. The whole process is graphically depicted in figure 1.

3.2 Profiling

Based on our previous work and an extended literature study, we identified a set of desired properties for components and microservices thus providing their profile. A profile is categorized into functional, non-functional and other properties:

(i) Functional properties: Include those properties that describe what the component or microservice actually does. It involves a general description of the functionality delivered through methods along with their specific descriptions.

(ii) Non-functional properties: Include properties reflecting how the component or microservice

behaves, mostly in terms of performance, using indicators such as time for execution, bytes of data processed per second, operations executed per second, number of concurrent users supported, cold start (the transition time from deployment to actual execution), etc.

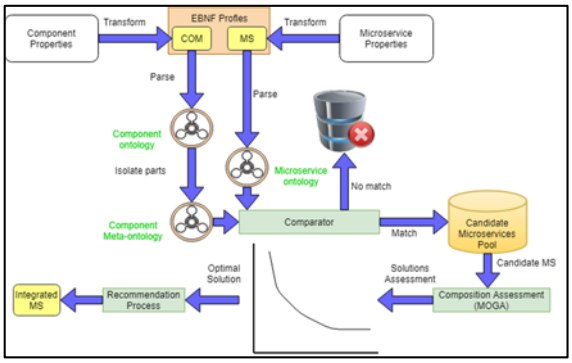

Figure 1: The proposed process for component decomposition and microservices substitution.

(iii) Other properties: Such properties involve other critical attributes of a component or microservice that may not be considered as functional or non-functional. These constitute mostly properties that provide useful general information regarding the artefact and its usage. In this part a profile provides information regarding the programming language it is implemented with, the level of security it provides, its auditability, data exchange, interaction protocol, type (data source, application login, GUI, etc.), data format, load balancing, obligations and constraints, automation and level of binding, verification and validation issues, cost, the data storage which describes how the component stores its data, and, lastly, a service descriptor.

Numerical properties included in the categories above may provide minimum or maximum threshold values, which will be used to guide the matching process for selecting suitable microservices for substitution.

As in our previous work we resort to using the Extended Backus-Naur Form (EBNF) to express component and microservice descriptions, which allows for formally proving key properties, such as well-formedness and closure, thus assisting in validating the semantics. The proposed grammar has been developed with the Another Tool for Language Recognition (ANTLR) (http://www.antlr.org/), a parser and translator generator tool that supports language grammars in EBNF syntax. Figures 2 and 3 depict the EBNF description of a component and a microservice respectively, which are analysed below.

The profile of a component includes, from top to bottom, the following: First, some definitions of component items are provided, including a name and a list of one or more services it offers. Each service is defined by a primary and a secondary function, the latter being more informative, as well as an optional description. Primary types involve general functionality, like I/O, security, networking, etc.; the secondary types explicitly define the actual function it executes, e.g. printing, authentication, video streaming, audio processing etc. For example, the service could be [Security, Login Authentication].

When decomposition takes place, this is one of the main features that will guide the searching for a microservice and it is considered as a Constraint, something which means that a candidate microservice will be rejected if it does not offer such functionality. Interfacing information comes next that outlines the various methods that implement its logic; a method is further analysed to Preconditions, Postconditions, Invariants and Exceptions, if any. This piece of information is provided upfront by the component developer/vendor. Non-functional requirements or properties are defined next denoting mandatory behaviour in terms of performance. Finally, general information intended to serve reusability purposes (application domain, programming language, OS etc.) is provided. It should also be mentioned that certain features in the profile may be assigned specific values along with a characterization as to whether this feature is minimised (i.e. the value denotes an upper acceptable threshold) or maximised (i.e. the value denotes a lower acceptable threshold) in the component under decomposition. For example, if performance is confined under 15 seconds, then next to the performance indicator the couple (15, minimise) is inserted.

The definition of the attributes included in the microservice profile is defined in a more detailed form compared to the component profile and thus the microservice profile may be considered as a more refined version of the component profile. The microservice profile first describes the data types used to define the property values. These types suggest three categories of microservice attributes. The first category is the ‘functional requirements’ where a specific textual description is provided regarding the function a microservice delivers. Since microservices are smaller and more specific than components, so is their description, which intuitively documents what it does. The second category is the ‘non-functional requirements’ which includes information describing mostly performance, as well as other constraints. These attributes include

performance indicators like bytes processed per second and operations executed per second, the level of security it provides, and information regarding its data storage (SQL, GraphDB, Document Store, File). The third and last category is 'other requirements' and provides additional general information regarding the microservice. The properties in this category include the programming language used to implement the microservice, the ability to audit events in logs (auditability), information regarding the data exchange protocol (REST, SOAP, RPC) and interaction protocol (Synchronous, Asynchronous) supported, data format in which data is exchanged (JSON, XML), load balancing, cost etc., as shown in figure 3.

After briefly describing both profiles, we will now focus on the process which connects components with microservices and demonstrate how component decomposition is performed and microservices are matched through ontology instances.

3.3 Components Decomposition and Microservices Matching

3.3.1 Component and Microservices Ontology

A special form of ontology is devised to facilitate the subsequent steps of decomposing a component into individual functional parts, and then locating and assessing the suitability of available microservices for integration using a self-contained description. The ontology is built around the property axes of the components and microservices profiles described above, the latter conforming to the same semantic rules as the former, so as to facilitate their automatic transformation to instances of the ontology. Figures 4 and 5 depict the largest parts of these ontologies, while some details have been intentionally omitted due to size limitations.

The matching process works at the level of the ontology tree and not the textual descriptions of the profile, something that makes comparisons more easy and quick, both computationally and graphically (visually). This is due to an ontology alignment algorithm used to compare (align) two same-structured ontology instances aiming at locating attribute similarities and attribute values distances. The ontology alignment algorithm works by parsing both ontologies as ontology tree instances and investigates their structure level by level in a tree structure hierarchy.

After the schema similarity is compared, the algorithm calculates the value distance for each attribute depending on their data type since there is a form of heterogeneity between attribute values. For example, some attributes may be of binary type, while some others may be of numerical. The solution is to handle distance calculation differently depending on the data type compared in each attribute. In our case the ontology alignment algorithm used is Graph Matching for Ontologies (GMO) (Hu et al., 2005). GMO initially parses two ontologies and transforms them to RDF bipartite graphs following some matrix operations to determine the structural similarity. This is the first and most crucial step of the proposed methodology as it initially discards non-matching microservices from the pool of available candidate microservices. The matching process is described in detail in the next section.

3.3.2 Matching Process

Different methods are proposed in literature for description processing, such as simple string, (Frappier, 1994), signature matching (Zaremski and Wing, 1993) and behavioural matching (Zaremski and Wing, 1997). The approach followed in this paper is slightly different; it employs a hybrid form combining string and behavioural matching. More specifically, a dedicated parser is implemented that recognises certain parts in a profile (functional, non-functional and other properties as previously described) which is translated into an ontology instance (either of a component or a microservice). The parser first verifies that the profile is expressed in the proper context and semantics of the structures presented earlier (see figures 2 and 3) using the ANTLR framework and then proceeds with building the ontology tree of instances according to the the recognized parts. Parsing and transformation essentially build the ontology tree instances that describe the software components under decomposition and the available microservices. The next step is to match properties between ontology items. The tree instance of the component under migration is projected on top of any other candidate microservice assessing the level of requirements fulfilment in two phases: The first phase checks that all required functions (the component's part to be replaced) are satisfied by the available microservices; therefore, we treat these as functional constraints.

In this case the list of services sought (decomposed part) must be at least a subset of the services offered (candidate microservices). The second phase is executed once all functional constraints are satisfied and calculates the level of suitability of each candidate microservice.

```
(**** Component EBNF Profile ****)
DIGIT : 0|1|2|3|4|5|6|7|8|9;
INTEGER : DIGIT {DIGIT};
CHAR : A|B|C|…|W|a|b|c|…|W|!|@|#|…;
STRING : CHAR {CHAR};
Variable_type : CHAR|INTEGER|…;
Variable_name : STRING;
Primary_Type : 'Input'|'Output'|'Security'|'Multimedia'|'Networking'|'GUI'|…;
Secondary_Type : 'Authentication'|'Data processing'|'Video'|'Audio'|'File access'|'Printing'|…;
Details_Description : CHAR {CHAR};
Min_Max_Type : 'Minimize'|'Maximize';
Required_Type : 'CONSTRAINT'|'DESIRED';
Service : 'S' INTEGER Primary_Type, Secondary_Type { Details_Description} Required_Type;
Service_List : Service {Service};
Operator : 'exists'|'implies'|'equals'|'greater than'|'less than'|…;
Condition : Variable_Name Operator {Value} {Variable};
Precondition : Condition {Condition}; (*IF THESE ARE PROVIDED BY DEVELOPER/VENDOR*)
Postcondition : Condition {Condition}; (*IF THESE ARE PROVIDED BY DEVELOPER/VENDOR*)
Invariants : Condition {Condition}; (*IF THESE ARE PROVIDED BY DEVELOPER/VENDOR*)
Exceptions : Condition {Details_Description} {Exceptions}; (*IF THESE ARE PROVIDED BY DEVELOPER/VENDOR*)
Method : 'M' INTEGER {Variable Variable_Type} {Precondition} {Postcondition} {Invariant} {Excpetion}; (*IF THESE
ARE PROVIDED BY COMPONENT DEVELOPER/VENDOR*)
(*==== INTERFACING ====*)
Service_analysis : 'Service' INTEGER ':' 'Method' INTEGER ':' STRING Method {Method};
(*==== NON_FUNCTIONAL PROPERTIES ====*)
Performance_indicators : [ 'Response time' (INTEGER) Min_Max_Type Required_Type | 'Concurrent users' (INTEGER)
Min_Max_Type Required_Type | 'Records accessed' (INTEGER) Min_Max_Type Required_Type | … ]
{Performance_indicators};
Resource_requirements : [ 'memory utiliztion' (INTEGER) Min_Max_Type Required_Type | 'CPU reqs' (INTEGER)
Min_Max_Type Required_Type | … ] {Resource_requirements};
Quality_features : [ 'Availability' (INTEGER) Min_Max_Type Required_Type | 'Reliability' (INTEGER) Min_Max_Type
Required_Type | … ] {Quality_features;
(*==== END OF NON-FUNCTIONAL PROPERTIES; NEW ITEMS MAY BE ADDED HERE ====*)
(*==== REUSABILITY PROPERTIES ====*)
Application_domain : 'Medical' Required_Type | 'Financial' Required_Type | 'Business' Required_Type | …
{Application_domain};
Programming_language : 'C' Required_Type | 'C++' Required_Type | 'Java' Required_Type | 'VB' Required_Type | …
{Application_domain};
Operating_systems : 'Windows' Required_Type | 'Linux' Required_Type | 'Unix' Required_Type | 'IOS' Required_Type |
'Android' Required_Type | … {Operating_systems};
Openness : 'black' Required_Type | 'glass' Required_Type | 'grey' Required_Type | 'white' Required_Type;
Price : INTEGER;
Development_info : STRING;
Developer: STRING;
Version : STRING; (*IF THESE ARE PROVIDED BY DEVELOPER/VENDOR*)
Protocols_Standards : [ 'JMS/Websphere' Required_Type | 'DDS/NDDS' Required_Type | 'COBRA/ACE TAO'
Required_Type | 'POSIX' Required_Type | 'SNMP' Required_Type | … ] {Protocols_Standards};
Documentation : [ 'Manuals' Required_Type | 'Test cases' Required_Type | … ]; (*IF THESE ARE PROVIDED BY
DEVELOPER/VENDOR*)
(*==== END OF REUSABILITY PROPERTIES; NEW ITEMS MAY BE ADDED HERE ====*)
SPECIFICATIONS PROFILE :
      'Specifications Profile :' STRING; 'Descriptive title :' STRING;
      'Functional Properties :' Service_List;
      'Interfacing :' Service_analysis {Service_analysis};
      'Non-functional Properties :' Performance_indicators, Resource_requirements, Quality_features;
        'Reusability Properties :' Application_domain, Programming_language, Operating_systems, Openness, Price,
                                   Protocols_Standards, Documentation;
```

Figure 2: Component profile in EBNF.

```
(**** Microservice EBNF Profile ****)

(*==== General Properties ====*)
BINARY : 'Yes' | 'No';
STRING : (' '..'~')+;
NUMBER: ('0'..'9')+;

WS : [ \r\n\t] + -> skip;
NEWLINE : [\r\n]+;

(*==== Functional Requirements ====*)
functional_description : STRING;

(*==== Non-functional Requirements ====*)
securityLevel : NUMBER;
bytesProcessedPerSecond : NUMBER;
operationsExecutedPerSecond : NUMBER;
coldStart : NUMBER '.' NUMBER;

(*==== Other ====*)
programmingLanguage : 'C' | 'C++' | 'Java' | 'Python';
dataStorage : 'None' | 'SQL' | 'Graph' | 'Document' | 'File';
auditability : BINARY;
dataExchange : 'REST' | 'SOAP' | 'RPC';
interactionProtocol : 'Synchronous' | 'Asynchronous';
type : 'Data Source' | 'Application Logic' | 'GUI';
dataFormat : 'JSON' | 'RSS' | 'XML';
loadBalancing : 'N/A' | NUMBER 'threads';
obligationsConstraints : 'Public' | 'Private' | 'Local';
automationLevelOfBinding : 'Manual' | 'Semi-automated' | 'Fully automated';
verificationValidation : 'Yes with test data' | 'Yes without test data' | 'No';
serviceDescriptor : 'N/A' | 'UML' | 'WSDL' | 'OWL-S' | 'BPEL';
cost : NUMBER '.' NUMBER;
```

Figure 3: Microservice profile in EBNF.

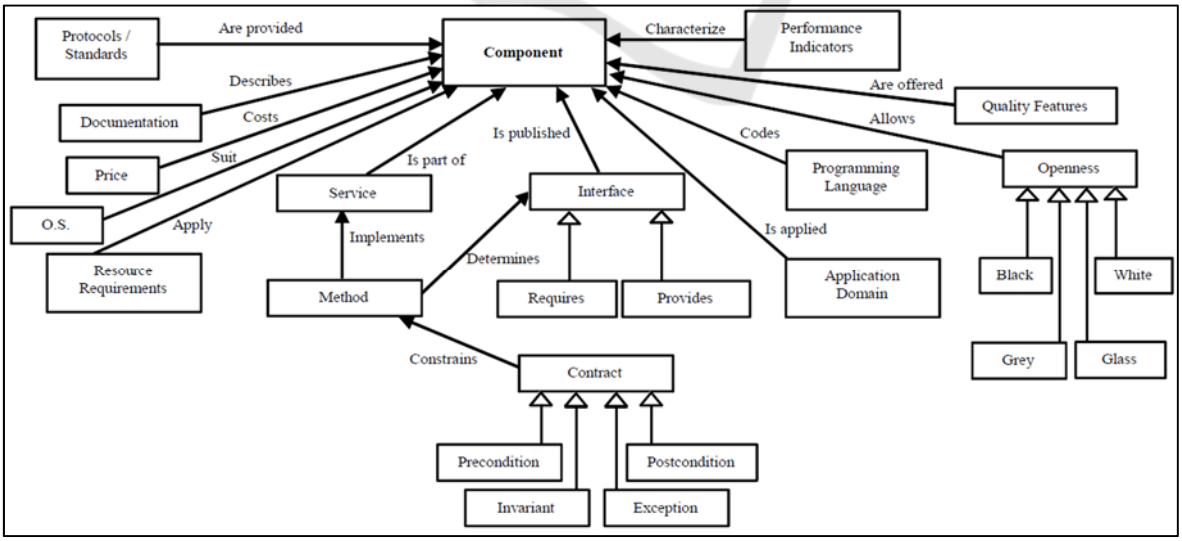

Figure 4: Component ontology.

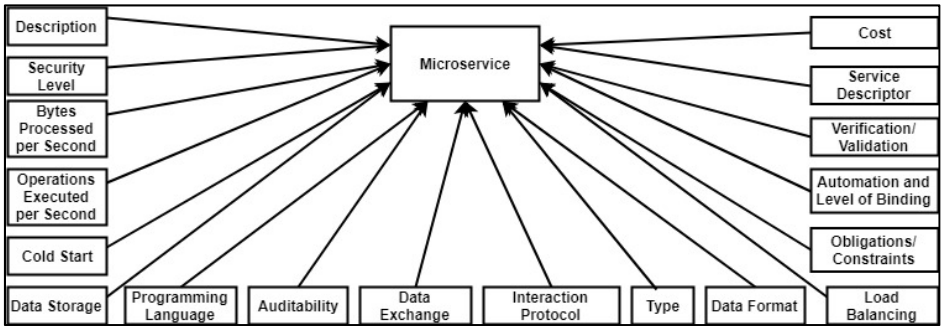

Figure 5: Microservice ontology.

A demonstration example for this phased approach is given in the experimental section, while a more detailed description of the matching process is provided below.

Firstly, the functionality offered by a software component is decomposed into one or more functions (methods) and the associated non-functional aspects (performance indicators). This is performed by traversing the ontology tree in a depth-first-search manner until we reach the leafs, that is, the details of the methods (e.g. interfaces, arguments, conditions, etc.) the component is made of (see component profile in figure 2). Then the algorithm climbs up the ontology structure until it reaches the definition of the method to which this detailed information refers. This way the functional parts of interest in the component's ontology instance are isolated creating a form of meta-ontology as depicted in figure 1 and described earlier. The non-functional properties are then visited on the ontology tree top-down using a string-matching approach, where we differentiate between two cases: (i) The overall performance indicator(s), which describe how the component behaves as one entity of integrated functions. This will be used during the synthesis part of the matching algorithm to guide the process of recommending microservices for integration taking into consideration how their combination should behave as a whole, any incompatibilities in terms of interfacing, timing (synchronous/asynchronous), its type (SOAP, REST), etc. (see experimental part in section 4); (ii) Method-specific indicators, that constrain the way a certain function (method) delivers its functionality as a single unit. This piece of information will be used by the matching algorithm when assessing the suitability of a microservice as it is considered a mandatory requirement. As soon as all meta-ontology parts (i.e. methods) are isolated, the proposed matching algorithm is invoked. Considering a single function (method) from the derived decomposition, we aim to match it with a candidate microservice that resides in the pool of available microservices. For simplicity, let the instance of the source (component) function for microservice substitution be denoted as M_{sk} (k=1..M, where M is the number of decomposed component functions), which is considered as the profiled microservice sought after decomposition (from now on we will refer to this as the 'source microservice'). At the other end, the profile of all of the available microservices is also parsed and ontology instances are created, let these be M_{ti} (i=1..N, where N is the number of available microservices). Due to the fact that there is a form of heterogeneity between microservice attributes that concern their data types, a combination of metrics is used in order to assess the matching score of each target microservice instance Ti while taking into account the aforementioned heterogeneity.

The ontology profile, as described through EBNF, has three distinct data types, namely binary, numerical and string. Therefore, a different metric function is used for each data type. For the binary data type, the similarity function score is given by the following formula:

$$s_{bin} = \frac{1}{N}\sum_{i=1}^{N} b_{st,i} \quad (1)$$

where

$$b_{st,i} = \begin{cases} 0, if\ binary\ attribute\ i\ required\ in\ M_s \\ is\ not\ satisfied\ in\ M_t \\ 1, if\ binary\ attribute\ i\ required\ in\ M_s \\ is\ satisfied\ in\ M_t \end{cases}$$

and M_s and M_t are the source and target microservice ontology instances respectively.

Respectively, the score between any two sets of numerical attributes is given by:

$$s_{num} = \frac{1}{N}\sum_{i=1}^{N} \{max_{st,i}, min_{st,i}\} \quad (2)$$

where $max_{st,i}$ is the formula for attribute i to be maximized between source and target ontology instances given by

$$max_{st,i} = 1 - \frac{n_{s,i} - n_{t,i}}{max(n_{s,i}, n_{t,i})} \quad (3)$$

and $min_{st,i}$ is the formula for attribute i to be minimized between source and target ontology instances given by

$$min_{st,i} = 1 + \frac{n_{s,i} - n_{t,i}}{max(n_{s,i}, n_{t,i})} \quad (4)$$

Since some attribute values can be maximized or minimized, we use the correct formula for attribute value similarity calculation each time. For example, the attribute bytes processed per second is maximized because it has to score higher if the value offered is higher than the desired one. On the contrary, the attribute cost has to be minimized due to the exact opposite reason. Cost similarity value has to score higher if the offered value is less than the desired one.

Lastly, the score between any two sets of string attributes s_s and s_t is given by the mean of the Jaccard similarity coefficient:

$$s_{str} = \overline{(J(s_s, s_t))} \quad (5)$$

where $J(s_s, s_t)$ is the Jaccard similarity coefficient between source and target string sets respectively, and is calculated as:

$$J(s_s, s_t) = \frac{|s_s \cap s_t|}{|s_s \cup s_t|} = \frac{|s_s \cap s_t|}{|s_s| + |s_t| - |s_s \cap s_t|} \quad (6)$$

where:

$|s_s|$ is the number of terms contained in string s_s, $|s_t|$ is the number of terms contained in string s_t, and $|s_s \cap s_t|$ is the number of shared terms between strings s_s and s_t respectively.

Using the equations above we can now describe the procedural flow of the matching algorithm. The algorithm consists of two sequential phases:

Phase 1: All attributes of the source microservice, which are considered as mandatory, must map one-on-one to the attributes of the target microservice. This means that, by traversing all of the available target microservices, each attribute of the source microservice is verified to exist in the target microservice. Otherwise, the target microservice is discarded and it is removed from the pool of candidate microservices. Therefore, after Phase 1 concludes, the pool of candidate target microservices has been reformed to include only those target microservices that in general match the mandatory requirements of the source microservice; the level of suitability of the microservices in this pool may vary depending on secondary, desired features or properties they may possess, the respective values of which are subsequently assessed in Phase 2 by the score functions previously described. As previously mentioned, Phase 1 is supported by a variation of the GMO algorithm which was developed to parse every pair of the compared microservice ontologies (source and target) and defines their structural similarity.

Phase 2: The similarity between a source microservice and a specific target microservice in the pool of candidate microservices formed by Phase 1 is assessed through the relevant score functions depending on their data type. The algorithm calculates the mean of binary, numerical and string score of the pair and produces a similarity value. This is repeated for every pair of source and target microservice in the pool, and the final outcome is a ranked matching score:

$$s_{tot} = \overline{(s_{bin}, s_{num}, s_{str})} \quad (7)$$

The matching algorithm is shown in figure 6.

4 EXPERIMENTAL PROCESS

A two-stage experimental process was designed and executed aiming to assess the efficiency of the proposed framework. Specifically, in the first stage (proof of concept) we examined the ability of the framework to deliver and recommend a list of microservices that are suitable to replace specific functions of a component, ranked based on the suitability score of equation (7). In the second stage (composition assessment) two MOGAS were employed to deliver near-optimal synthesis of candidate microservices taking into account the required dependencies as these were defined in the software component design. All scripts that support the aforementioned experimental environment were implemented in Python 3.7 and the full sets of results are available in this link [1]. The two stages are described in detail below:

4.1 Proof of Concept

During the first stage of the experimental evaluation, we have tested the proposed framework using two specific cases. In the first case we consider having the functional parts (profiles) of a decomposed CRUD

[1] https://tinyurl.com/y8deeffz

component, which provides the simple functions of create, read, update and delete for a business artefact (e.g. a customer or invoice). We assume in this case that there are no dependencies across the functional parts of the decomposed component, that is, every operation is individual and does not depend on any other operation. This means that execution of one of the above operations does not require the prior execution of another. In the second case we focus on seeking to replace the functional parts of a component that are part of an inventory system and are dedicated for invoice updating. This component consists of five different functions as follows: *Update Invoice Items*, *Update Invoice Headers*, *Update Corresponded Posting*, *Update Debtor's Balance* and *Print Invoice*. These functions are all sequentially dependent (in the order listed), that is, every function depends on its previous one and starts as soon as its predecessor has concluded.

For the execution of the experiments, two EBNF profiles, aligned with our proposed framework, were created so as to fulfil the description of the two software components in hand for the stages described above. A pool of 5000 synthetic microservices profiles were randomly constructed ensuring that a minimum number of 200 microservices match the requirements of each functional part for both software components. This intuitively means that we make sure that every decomposed part has at least 200 candidate microservices in the pool that are matched and satisfy the mandatory requirements but with unknown suitability score. This will enable examining the correctness of our matching algorithm. We validated the scores computed by the matching algorithm by varying certain attribute values of the functional parts that derive from the decomposed component and repeating the matching process. We observed that by varying the attribute values in a series of repetitions and experiments, the matching algorithm correctly yields different scores and proper rankings among the candidate microservices as expected. This verified that changing the requirements of the functional parts triggers different scores and microservices that previously matched a specific functional part with a relatively high score tend to score lower when the requirements shift and vice versa.

4.2 Composition Assessment

As explained above, this experimental stage aims to examine the suitability of the utilization of heuristic approaches to deliver near-optimal microservices synthesis considering the satisfaction of two or more objectives related to non-functional characteristics of the software component. The vast solution space of the problem under study prohibits the utilization of computational process. We resorted to using heuristic

```
#Decompose software component
source_microservices = decompose(component)

#Parse ontologies
for s in source_microservices:
 M_s = parseOntology(s)
 for M_t in target_ontologies:
       #Phase 1
       candidates = []
            #Structurally similar microservices cause the target microservice to be included in the
            microservice candidate pool
       if(GMO(M_s, M_t) == 1):
            #Phase 2
            candidates.append({M_t : score(M_s, M_t)})

#The score function is the algorithms' Phase 2 which is implemented below according to the similarity
functions as defined above:
def score(M_s, M_t):
 s_bin = scoreBinary(M_s.getBinaryAttributes(), M_t.getBinaryAttributes())
 s_num= scoreNumerical(M_s.getNumericalAttributes(), M_t.getNumericalAttributes())
 s_str = scoreString(M_s.getStringAttributes(), M_t.getStringAttributes())
 score = (s_bin + s_num + s_str) / 3
 return score
```

Figure 6: Microservice ontology matching algorithm.

approaches and, more specifically, genetic algorithms, as our problem was rich in candidate solutions with conflicting objectives; therefore, we selected multi-objective genetic optimization as it has been proven to be quite efficient in such cases.

Two MOGAS were selected to solve the multi-objective optimization problem, which will also be used to compare their performance and effectiveness: The Non-dominated Sorting Genetic Algorithm II (NSGA-II) and the Strength Pareto Evolutionary Algorithm 2 (SPEA2). The selection of these two specific algorithms was made due to their wide acceptance and use, but most importantly their good performance in such kind of applications which was proven in our case too after a quick verification with preliminary runs. The multi-objective optimization environment was accordingly adjusted and configured based on the problem under study. The two objectives formed are the minimization of the microservice cost and the execution time (performance) respectively. We assume that the two are competing in the sense that the higher the performance the more expensive the microservice. The set of decision variables was constructed by five vectors each corresponding to a decomposed function and yielding values related to the selected candidate microservice that delivers the same functionality. Two constraints were also set, one for each objective, both denoting an upper value for the objectives (cost, time) that cannot be tampered. The experimental implementation of the algorithms was performed using Platypus [2], a Python-based multi-objective optimization algorithms library.

4.3 Results and Discussion

The results generated by the execution of the proposed process over the two experimental cases are provided in Tables 1 and 2 respectively (sample of the best five ranked microservices). The results consist of the id of the best five microservices for each functional part along with their matched score in descending order.

Table 1: Scoring results of components' functional parts without dependencies.

Create	Read	Update	Delete
184 (0.48)	1316 (0.62)	445 (0.89)	1317 (0.75)
37 (0.48)	227 (0.56)	406 (0.89)	814 (0.63)
91 (0.47)	353 (0.54)	524 (0.87)	747 (0.55)
53 (0.44)	236 (0. 53)	409 (0.87)	659 (0.53)
73 (0.44)	379 (0.51)	563 (0.86)	728 (0.52)

[2] https://platypus.readthedocs.io

Table 2: Scoring results of components' functional parts with dependencies.

Print invoice	Update invoice items	Update invoice headers	Update debtors balance	Update posting
800 (0.65)	104 (0.50)	329 (0.63)	740 (0.61)	421 (0.90)
1455 (0.60)	1506 (0.49)	1612 (0.60)	692 (0.59)	545 (0.89)
1995 (0.58)	65 (0.48)	312 (0.55)	706 (0.58)	499 (0.89)
2079 (0.57)	101 (0.47)	273 (0.53)	611 (0.55)	524 (0.88)
2137 (0.57)	82 (0.47)	231 (0.52)	1506 (0.54)	1480 (0.87)

For the first case, a total number of 955 unique microservices have been positively assessed and included in the candidate microservices pool in descending order based on the calculated suitability score. Specifically, Create function included 249 candidates, Read function 225 candidates, Update function 233 candidates and finally Delete function 248 candidates. As regards the second case, a total number of 1709 unique microservices have fulfilled the mandatory requirements and were selected to be included in the candidate microservices pool as follows: 652 microservices were included in Print Invoice function's list, 283 in Update Invoice function's list, 236 microservices in Update Invoice Headers function's list, 280 microservices in Update Debtors Balance function's list, and, finally, 258 microservices are included in Update Posting function's list.

Firstly, we observed that our algorithm performed successfully discarding all candidate microservices that failed to satisfy even one mandatory requirement. Secondly, by choosing and comparing arbitrarily microservices from the same list of candidates we confirmed the correct assessment of the microservices by the matching algorithm reflected in the calculated suitability scores, as well as the correctness of their prioritization.

As described in the experimental process design, the results extracted from the second case (software component decomposed into a series of dependent actions), were then used for the assessment of the microservices synthesis. The number of possible solutions (PS) in this case is calculated by equation (8) to be over 3 trillions.

$$|PS| = \prod_{i=1}^{N} x_i \quad (8)$$

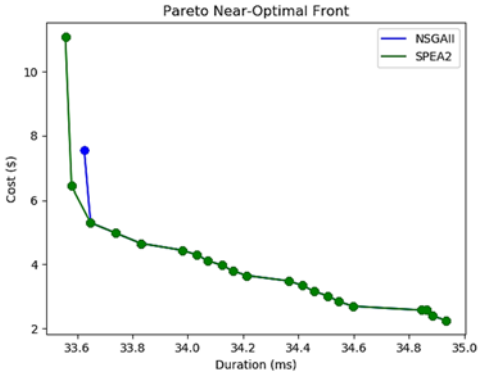

Figure 7: Near-optimal Pareto fronts.

N in eq. (8) corresponds to the number of decomposed functions and X_i is the number of recommended microservices for function i.

Each MOGA was run 100 times for 500000 fitness evaluations (FE) resulting in the generation of 100 Pareto fronts. By combining these Pareto fronts, a near-optimal Pareto front was produced for each algorithm. The two near-optimal Pareto fronts are depicted in figure 7. The first observation one can make when inspecting the Pareto fronts is that both MOGAs delivered similar solutions. Going a step further and by studying the microservices combinations which corresponded to the optimal solutions, we observed that microservices with high individual suitability scores were missing from the proposed optimal solutions and respectively microservices belonging to the optimal solutions sets had relatively low suitability scores compared to others. This finding is perfectly reasonable as the specific experiment was focused on optimising cost and performance, while the suitability score is the collection of other parameters as well. Therefore, the recommended solutions that will drive the synthesis of microservices will always depend on the aspects designers need to optimise each time.

The performance of the two MOGAs was assessed and compared with the use of the Hypervolume (HV) (Thiele and Zitzler, 1999) and the Inverted Generational Distance (IGD) (Veldhuizen and Lamont, 2000) quality indicators. The HV indicator assesses the volume covered by the non-dominated solutions of a Pareto front in the objective space and therefore, the larger the volume covered by the solutions generated in a run, the higher the HV value, which indicates a better performance. The IGD indicator assesses how far the elements of the true Pareto front are from the non-dominated points of an approximation Pareto front and therefore, the greater the extent of the true Pareto front that is covered by the non-dominated points generated by a run in the objective space, the lower the IGD value, which denotes a better performance. Each algorithm was run 10 times and both HV and IGD values were calculated for each algorithm. In order to compare the performance of the two algorithms the median HV and IGD were calculated. The HV value for both algorithms was identical and equal to 0.0257. The IGD value for the NSGA-II was 0.6113 and for SPEA2 0.6150.

Considering that the results of the two indicators suggest a balanced performance with no clear distinction being observed between the two algorithms used, we may safely conclude that none of the two overcomes the other. Two statistical tests were used to determine if there is any statistical difference between the two algorithms. Both the Wilcoxon signed-rank test and the Mann-Whitney U test suggested that there is no statistical difference ($p<0.05$) between the HV and IDG results of the two algorithms. Therefore, the two MOGAs are equally suitable to offer a sound basis for automatically guided microservices synthesis.

5 CONCLUSIONS

While microservices architecture is gaining wide adoption in the software development process, the contribution of this research work is to support software developers migrate from software components to microservices. Guided by three research questions, this paper aims to provide a well-described automatic process that identifies and recommends the full or partial replacement of a software component's functionality by a number of available microservices. The proposed process comprises a series of tasks which a developer may follow to receive a recommended solution. The component is expressed in a semi-formal notation in EBNF which is parsed to identify its functional parts. This identification takes place using an ontology scheme. The decomposed functions are then matched against available microservices. First, the microservices are screened based on the required functionality and the successful candidates are scored using a matching algorithm. Additionally, the proposed process is integrated with search-based techniques and recommends the optimal synthesis of microservices yielded by Multi-Objective Genetic Algorithms. The proposed process was evaluated through a two stage experimental process and presented successful performance in delivering proper solutions.

Quite a few challenges and open issues exist on the specific topic, some of which constitute our future work. Specifically, the constant increase in the availability of microservices with business orientation will require the design and execution of more advanced and extended experiments. Furthermore, an investigation will be performed for improving the profiling tasks by adopting different description models and assess whether this may improve also the automation level of the proposed process. Finally, more real-world cases will be employed to assess further the practical benefits of our approach.

ACKNOWLEDGEMENTS

This paper is part of the outcomes of the Twinning project Dossier-Cloud. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 692251.

REFERENCES

Andreou, A. S. and Papatheocharous, E. (2015) Automatic matching of software component requirements using semi-formal specifications and a CBSE ontology, in *Evaluation of Novel Approaches to Software Engineering (ENASE), 2015 International Conference on*, pp. 118–128.

Barba, L. A. (2005) Computing high-Reynolds number vortical flows: A highly accurate method with a fully meshless formulation, in *Parallel Computational Fluid Dynamics 2004: Multidisciplinary Applications*. Springer, Cham, pp. 305–312.

Baresi, L., Garriga, M. and De Renzis, A. (2017) Microservices Identification Through Interface Analysis, in *Service-Oriented and Cloud Computing*, pp. 19–33.

Cai, X., Lyu, M. R. and Wong, K. (2000) Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes, in *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*. IEEE Comput. Soc, pp. 372–379.

Dragoni, N. *et al.* (2017) Microservices: Yesterday, Today, and Tomorrow, in *Present and Ulterior Software Engineering*. Cham: Springer International Publishing, pp. 195–216.

Esposito, C., Castiglione, A. and Choo, K.-K. R. (2016) Challenges in Delivering Software in the Cloud as Microservices, *IEEE Cloud Computing*, 3(5), pp. 10–14.

Frappier, M. (1994) *Software Metrics for Predicting Maintainability Software Metrics Study : Technical Memorandum 2*, *Source*.

Gysel, M. *et al.* (2016) Service Cutter: A Systematic Approach to Service Decomposition, in. Springer, Cham, pp. 185–200.

Hu, W. *et al.* (2005) GMO: A graph matching for ontologies, in *Proceedings of K-CAP Workshop on Integrating Ontologies*, pp. 41–48.

Jatoth, C., Gangadharan, G. R. and Buyya, R. (2015) Computational intelligence based QoS-aware web service composition: A systematic literature review, *IEEE Transactions on Services Computing*, 10(3), pp. 475–492.

Kleiner, A. (2009) Making It Easy to Do the Right Thing, *IEEE Software*, 33(3), pp. 53–59.

Levcovitz, A., Terra, R. and Valente, M. T. (2016) Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems.

Moghaddam, M. and Davis, J. G. (2015) Service Selection in Web Service Composition : A Comparative Review of Existing Approaches, in *Web Services Foundations*. New York, NY: Springer New York, pp. 321–346.

Rosen, M., Lublinsky, B., Smith, K., and Balcer, J. (2008) *Applied SOA: Service-Oriented Architecture and Design Strategies*. Wiley.

Thiele, L. and Zitzler, E. (1999) Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach, *IEEE Transactions on Evolutionary Computation*, 3(4), pp. 257–271.

Veldhuizen, D. A. Van and Lamont, G. B. (2000) Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art, *Evolutionary Computation*. MIT Press, 8(2), pp. 125–147.

Zaremski, A. M. and Wing, J. M. (1993) Signature matching, in *ACM SIGSOFT Software Engineering Notes*, pp. 182–190.

Zaremski, A. M. and Wing, J. M. (1997) Specification matching of software components, *ACM Transactions on Software Engineering and Methodology*. ACM, 6(4), pp. 333–369.

Zeginis, C. and Plexousakis, D. (2010) *Web Service Adaptation: State of the art and Research Challenges*, *Cycle*.