Karlsruhe Institute of Technology

Software Design and Quality

From Monolith to Microservices: A not yet defined Approach

Bachelor's Thesis of

Niko Benkler

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. B
Advisor:	Dr. Robert Heinrich

xx. Month 20XX – xx. Month 20XX

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . .

(Niko Benkler)

Bla Sbtrakt

Abstract

English abstract.

Zusammenfassung

Deutsche Zusammenfassung

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

1.2. Problem Statement

1.3. Challenges

2. CoCoME

2.1. Introduction to CoCoME

[7]

3. State of the Art

3.1. Literature Review

Link	Titel	Author (Year)	Origin	Search String
[6]	Extraction of Microservices from Monolithic Software Architectures	G. Matzlami et. al. (2017)	Google Scholar	*microservice identification*
[1]	Object-Aware Identification of Microservice	M. J. Amiri (2018)	IEEE	*identification microservices*
[2]	Microservices Identification Through Interface Analysis	L. Baresi et. al. (2017)	google scholar	*microservice identification*
[9]	Identifying Microservices Using Functional Decomposition	S. Tyszberowicz et. al. (2018)	*provided*	*n/a*
[8]	Partitioning Microservices: A Domain Engineering Approach	I. J. Munezero et. al. (2018)	IEEE	*identify microservices*
[3]	From Monolith to Microservices: A Dataflow-Driven Approach	R.Chen et. al	IEEE	monolith to microservice
[4]	Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems	A. De Alwis et. al. (2018)	Google Scholar	identify microservices
[5]	Service Cutter: A Systematic Approach to Service Decomposition	M. Gysel et. al. (2016)	[2]	*n/a*

3.2. Comparison and applicability of the approaches

* bedeuted INFO
+ beudeted PRO
- bedeuted CONTRA

3.2.1. Extraction of Microservices from Monolithic Software Architectures

* informal migration patterns exists. Lack of Formal Models
* small and recent body of work on how to migrate monolith to MS
* clas based extraction model, construct graph, process by clustering algorithm
* references Service Cutter (Pros and Cons)
* 2 phases: Construction (monolith to graph), clustering (decompose graph to cluster)
* starts with code base/repo from VCS
* each class is a node, edges have weights according to coupling strategy (classes that are not coupled are discarded)
* Logical Coupling Strategy(LC): Single Responsibility principle (Software has only one reason to change), enforce strong module boundaries (concept of MS) –> developers only make changes to the module (found in Change History, Class Files changed together belong togehter) ==> Weight is : for each pair of class look how often they changed together
* Semantic Coupling Strategy(SC): each MS correspond to one bounded context (DDD) from domain, examine contents/semantics of source code, term-frequency invese-documents-frequency method (tf-idf), compute relation of two classes regarding domain concepts
* tf-idf: Compute scalar vector for each class and compute cosine similarity between pairwise distinct classes
* tf-idf: Tokenize class, set of words, filter stop words, compare two classes regarding their common words with tf-idf formula
* Main Concern: Well organized teams, cross-functional but also reduce communication overhead to external teams while maximize internal
* Contributor Coupling (CC): team/orga info used to recover relationship among sw artifacts ==> Ownership architecture read from VSC history by identifying how many developers worked on the same pair of classes (weight!)

* CLustering Algo: Invert weights to favor edges with heigh weight, Kruskal for MST (calculate remaining edges), sort edges, reverse list, delete first element (originally lowest weight; now first in the list), in each iteration step: DFS(Depth-First) on edgesMST returns number of partitions, iterate and delete while n<nGiven *Reduce Cluster: after n cluster were formed, method reduceClusters splits up unusually large clusters (due to language and framework, special classes may have extraordinary high coupling) –> Delete those classes until size of node (numbe rof classes seems to be appropriate) ————————————

+ algorithmic recommendation of ms candidates implemented in web-based prototype
+ unites traditional decomposition techniques and microservice extraction approaches/design principles
+ algorithm uses 3 different coupling strategies: Can be combined for better results
+ performance of algortihm was satisfying according to author + shows significant team size reduction (less than half), given that alls contributirs for given cluster work on a MS
+ no work in advance!!!!

——————————————

- rely on (meta-)data extracted from codebase
- needs VCS (proper change history)
- needs ORM model that models data entities as ordinary classes - 2 (independent) changes in one commit destroy SRP
- SW must have gone through evolution process for LC
- Naming of class, methods, attributes needs to reflect domain language to make tf-idf possible
- Contributor Coupling: requires more developers
- Reduce Clusters: Useful to delete those classes? wat if this is core of application that belongs together - nPart is given by user: What is the right granularity?

——————————————

*Conclusion: Beta implementation existing, paper is precise in usage, master-thesis exists (LINK!!!)with detailed information, easy algorithms, good performance,
* No Information about right n, not domain oriented, ORM necessary, BUT: Not applicable to CoCoME as VCS history, software evolution, several contributor necessary

3.2.2. Partitioning Microservices:A Domain ENgineering Approach

* One BusinessCapability determines one MS, based on domain engineering * Business Capability: Something hat a business does;combined as functionality that have something in common (instead of usin collections of data entities and CRUD operations in them) * Paper states: Developers struggle to define granulaity of business capability (too small => Communication overhead, to large: heavyweight SOA) * Appropriate MS size determined by component boundaries * DDD: Choosing appropriate boundary. MAIN GOAL: systematically group requirements in domain model and implement code of that * Design Domain model with sub-domains (each sub-domain has one to more bounded contexts) (* Sub-Domain is problem space, bounded.context is solution space and maps software artifacts to sub-domains ==> Info aus Internet) * DDD Patterns: Context Map/Counded Contexts (makes explicit boundaries between domains,), Aggregates (logical boundaries for cluster of domain objects that change during one transaction, ensure consistency among parallel operations that are considered to be one unit in reagrd to change), Ubiquitous Language (ensures that impementation uses the same terms as business), Separation of Entities and Value types (Entities are Objects of DDD, Not all Objects need to be entitites, for value objects identity is not necessary)) * MS should be autonomous (changes not affecting others): One bounded context is one MS (ore more if fine grained needed, BUT never on C in different MS) * Approach: Defined domain and ubiquitous language ar prerequisites, split into sb domains if domain is too wide, find boundary of each responsibility, make it as business capability (focus on relationship among different services), each capability is a MS, find out relationships between services with domain model and reconfigure if necessary (too many dependencies)

————————————————

+uses DDD. Many papers and experience reports state that this is the way to gto

——————————————

- draw the boundaries is the ky task and requires domain experts - requires well defined domain, pre-existing ubiquitous language - "Find the boundary of each responsibility and make it as a business capability" ==> But how? Requires DOmain experts - Procedure is not concise at all and only explains approach on a top level (half a page) - Does not define how procedure analyses "loose coupling possible" - how break down in subdomains? - uses brainstorming and interaction wit domain experts to identify and define parts of the microservice

——————————————

* Summary:
*Hot topic according to other papers (interviews, reports) but this paper lacks on accuracy.
* Simply said: Get some domain-experts and let them cut your domain in different bounded contexts
* This is what was already done when migrating CoCoME (probably not following the patterns but having domain experts)

3.2.3. From Monolith to Microservices: A Dataflow-Driven Approah

*top-down dataflow driven decomposition algorithm * 3 step approach
* purified DFD (focus on data's semanteme and operations only, excludes side information), more data focused than traditional
* purified DFD (PFD) is a directed Graph with nodes (processing operations nodes and data nodes) and Approach:
* construct traditional DFD (according to business logic extracted from users' natural-language descriptions)
* Manual construction of purified dataflow by two rules (more like guidelines)
*algorithmic construction (condensing of PDS) of the decomposable Dataflow by applying a set of rules * combines same operations with same output data * each operation at the end is a MS

—————————————————

+systematic Algorithm + purified DFD represents real information flow of corresponding business logic + reduce human mistakes when drawing a composable DFD + the 5 rules are easy to understand

—————————————————

- Semi Automated
- traditional DFD constructed on users descriptions and code (detailed data flow) ==> exact? Expertise necessary
- or: DFD needs to be existant
- transforming traditional to purified is a non trivial task
- identifying same data operation requires expertise: Combination of sam operations based on operation names (what if names are different through the project) - no implementation

- not applied to larger projects - based on user's natural language: semantic verbs correspond activities (later a MS)
-really really fine-grained MS, based on single operation (most fine graines as possible)
- no domain information included
- what about get/create.... hier weiß ich nicht, wie ich beschreiben soll dass es nicht geht!
- zweites beispiel zigt das deutlicher mit der operation QUERY

——————————————————

Conclusion

TODO: Why not applicable to CoCoME

3.2.4. Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems

* function splitting based on i) Object subtypes (lowest granularity of sw based on behavioural properties) and ii) common execution fragments across software (lowest ... based on structural properties)
* authors say: MS not adopted for enterprise systems (ERP/CRM Sw which is large and has complex business process)
* lot of BO's (business objects) with many-to-many relationship, functional dependencies
* key strumbling block: "limited insight from syntactic structures of code for profiling software dependencies and identifying the semantics available through the BO relationships. * Process:
* ES system with modules that have function which execute basic CRUD ops on BOs (centralized DB) vs. own DB in MS (structural difference) * Behavioural property: Based on invocation of properties in well-defined processing sequences (different execution sequences reflect a set of SESE regions single-entry single-exit)
*Enterprise System as finite automaton with operations as labels: Vertices defined by states and relation defined by transistion functions
* SESE-Fragments: Induces a function (SubGraph of ES), Function is set of operations
* behaviour (of both MS and ES) based on invocation of operations –> Analyse those processng sequences
*Heuristics:
* given: several similar Execution paths with changed BOs, only Attributes they use are different (structural splitting of objects at BO level) * i) Subtype Relation in SESE(S): Subtype between callgraphs exists if ops, input/output of child are present in parent AND 80percent of parent states appear in child
* given: execution pattern occurs in several patterns (depend on functional relationship): A always before B ==> "as a relationship" property
* ii) Commonality: two call graphs have common sub-graph ==> Small Subgraphs (small MS), large Subgrapgh (large MS)
*Process:

* two components: Business Object Analyser (BOA), System Dynamic Analyser(SDA)
* BOA: Two models, one for evaluating the SQL queries to identify relationships between DB tables and one for identifying BOs based on the relationships and data similarities ==> Input for SDA
*SDA: BOs and callgraphs as Input, identifies Frequent Execution Patterns(FEP) in provided SESEs(S) by using clustering algorithms
*FEP is evaluated against heuristics and classified in categories
* categorized patterns evaluated by BO Relationship Analysis and SESE derivation model
* Microservice Recommendation Interface (MRI) provides MS recommendations
* SDA+MRI: two algos i) set of subgraphs in given set of callgraphs ii) analyse subgraphs for SESE code fragments (functions) that are related to same BO

——————————————————

+ prototype implementation exists
+ already conducted two experiments

——————————————————

- incremental process: most prominent components extracted and reimplemented as MS
- really complex
- BOA not given in paper
- uses a lot of algo that are not explained
- First experiment: log data analysed with DISCO to generate call graphs

——————————————————

Coclusion:
* Approach not fully implemented, implementation not available
* Complex algorithms that use other algorithms from other papers
* log data to get call graphs
* We do not want to incrementally extract MS
* Experiments focused on speed/ressources/time per requests ==> Did not focus on finding the "right" MS according to MS principles
* Not applicable to CoCoME

3.2.5. Identifying Microservices using functional decomposition

* based on UC specifications of sw requirements and functional decomposition of those requirements
create model of the system: finite set of system operations (public methods as response to external triggers) and system state space (system variables written/read by operations)
*system decomposition: each MS has own state space and operations (Goal are disjoint sets, read/write of others state space only via API)) –>syntactical partition of the state space
*system requirements: give by natural language/formal models/existing implementation

*functional requirements: use cases describe how users interact with system
* system operations/ system state variable extracted of Use cases –> operation/relation table
* operation/relation table: relationship between each op and state variable this op reads/writes
* extracting was done using text analysis tools
* Source COde Clustering is possible but only to help to understand structure of the system (if design is bad, the resulting clusters might not be appropriate MSs)

*Approach:
* first approximation: verbs in UC are operations, nouns are state variables (list might be updated, see page 5)
* generate operation/relation table
* Visualize op/rel table as graph, create cluster (good ms candidate low coupling (small amount of information sharing with others) and strong cohesion criteria(internal relationships are dense)!)
* Graph: vertices are state variable AND operations, edge between them if read/update, weights (1 for read, 2 for write)
* Use Graph Analyse Tools to determine clusters and read the MS (cluster = MS)
* Create API (all public ops in the cluster) and own DB for each MS
*Add Getter to API if other cluster needs information of this cluster *

——————————————————

+ approach is universally applicable once ops and state variable are identified (do not need to be extracted from use cases)
+ 3 ways to use the visualization: separation in low coupled partitions, partitioning in MS by non-functional constraints, visualize changes (re-engineering sw architecture)
+ identify MS based on business capability is main goal, not reducing MS size
+ 3 evaluation by three independent sw implementations: really similar to this approach
+ manual approach was matter of days, this approach only matter of hours

——————————————————

- synonymous/irrelevant nouns need to be identified via brainstorming –> User might be biased
- no security concerns
- no differences between different reads/writes (always 1/2)
- difference between read or write if other service? same value as REST time dominates

——————————————————

Conclusion:
* already done to CoCoME with good results.
* Hard to find substantial improvements?!

3.2.6. Microservice Identification Through Interface Analysis

* based on semantic similarity of available functionality described through OpenApi Specification and reference vocabulary –> collocation and similar words (DISCO)
*Idea: Same reference concepts are highly cohesive (Level in Hierachy of Vocabulary determines Granularity)
* match terms used in OpenAPI specification (as input) against reference vocabulary
* this is done iteratively on concepts by means of fitness function based on semantic similarity (DISCO)
* co-occurence matrix that contains all mappings of possible pairs of terms and concepts

* Process:
* for each api specification: Map each operation to concept that describes the operation most accurately
* Disco-based semantic assessment: Each operation along with resources (parameters, return values, complex types) is analysed and mapped to reference vocabulary concept using a score (Best Mapping for of term list and all concept available)
* Uses term separator (robust according to author), filters stop words, split word in input terms
* score based on fitness function: all collocation scores (between different words in term list and concept) stored in Matrix
* Mapping is non-trivial, use Hungarian algorithm to determine most adequate mapping
* Concept with highest mapping score is elected as reference concept

——————————————————

+ automated apart from defining OpenApi
+ according to authors: shared vocabulary helps to identify certain concepts with different meanings across system
+ APIMatic Tool can generate OpenApi sepcifications from existing interface
+Level parameter to define granularity of groupings (= MS)
+ implementation available

——————————————————

- Needs OpenApi Specification, re-engineering of available interfaces or new interface definition based on available artifacts
- based on reference vocabulary: They use Schema.org (too wide) –> Any other shared vocabulary ir own ontology needs to be constructed/used)
- Mapping to operation highly depends on used vocabulary
- Api needs well defined (provide proper naming) –> Might happen that two different operations (from two different sub-domains accoding to DDD have same naming).
- Reference vocabulary need to be tree
- Schema.org seems to be way to coarse-grained for CoCoME
- define own ontology probably ends in more work than manually defining bounded contexts
- lots of Intangibles (Unbestimmbar in Schema.org)

- Vocabulary might be misleading. From DDD point of view, similar words according to Disco belong not automatically to same business capability

——————————————————

Conclusion:
*Not applicable to CoCoME mainly because of Schema.org (for example: Sell Action and OrderAction in same concept)
*Concerns about well defined API * No idea how to adapt it

3.2.7. Service Cutter

* service decomposition based on 16 coupling criteria (coming from industry and literature) * service requires resources: Data (should only managed by one service), operations(service encapsulates business rules), artifacts(snapshot of data or operation results transformed into specific format) –> Geneeralize under term nanoentity *Approach: Identifying set of services and assign all nanoentities (each) to one service *Coupling criteria: architecturally significant requirements and arguments why two nanoentities should or should not be owned/exposed by same service *Software System Artifacts(SSAs): design/analysis atifacts that contain information about coupling criteria * service cut: output of single execution of service decomposition process * coupling criteria catalogue: <some examples>..., 4 categories: cohesiveness, compatibility, constraints, communication * structures using coupling criterion cards in specific format: ubiquitous language to structure, identify decisions

*Process: * Input in form of SSAs (Use Cases, DDD entities/aggregation, Entity relationship models) . Service Cutter extracts coupling criteria information of them * various additional SSAs * Service Cutter creates nanoentities(coupling criteria based on SSAs) * User priorizes these criteria (start calulation) * SC creates unidirected weighted graph with nodes that represent nanoentities and weighted edges that represent cohesiveness/coupling of two nanoentities * Clustering Algorithm (Can be changed!) * Weights: Sum of all scores (5 different types of scores) multiplied by priority
——————————————————

+ cited by many papers + catalogue consists of DDD pat7terns but also software quality attributes for architecture + Clustering Algorithm can be changed by user if necessary + user can prioritize coupling criteria (needs user experience, otherwise might result in wrong clustering) according to whats important for users use case (security yes or no...) + implementation available that was used by other papers to compare their approach + universally applicable to all software systems + wiki exists + can be used in forward and backward engineering
——————————————————

- not fully automated –> Only support - user-prioritized coupling criteria (well can also be positive) - lots of user input, needs to be in specific format (work intense!) - results and other papers show that SC does not "cut" the best solution

——————————————————

Conclusion:
* First attempt to automatize service extraction * To much effort for input (for bigger systems) * Already implementation and good approach (Further improvement possible?) * Good to compare with own approach * Nice tool

4. Solution Overview

5. Evaluation Planning

5.1. Applicability to CoCoME

5.2. Comparison to Functional Decomposition Approach

6. Timetable

6.1. Milestones

Bibliography

[1] M. J. Amiri. “Object-Aware Identification of Microservices”. In: (July 2018), pp. 253–256. ISSN: 2474-2473. DOI: 10.1109/SCC.2018.00042.

[2] Luciano Baresi, Martin Garriga, and Alan De Renzis. “Microservices Identification Through Interface Analysis”. In: (2017). Ed. by Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen, pp. 19–33.

[3] R. Chen, S. Li, and Z. Li. “From Monolith to Microservices: A Dataflow-Driven Approach”. In: (Dec. 2017), pp. 466–475. DOI: 10.1109/APSEC.2017.53.

[4] Adambarage Anuruddha Chathuranga De Alwis et al. “Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems”. In: (2018). Ed. by Claus Pahl et al., pp. 37–53.

[5] Michael Gysel et al. “Service Cutter: A Systematic Approach to Service Decomposition”. In: (2016). Ed. by Marco Aiello et al., pp. 185–200.

[6] G. Mazlami, J. Cito, and P. Leitner. “Extraction of Microservices from Monolithic Software Architectures”. In: (June 2017), pp. 524–531.

[7] Frank Mittelbach. “How to influence the position of float environments like figure and table in LaTeX?” In: *TUGboat* 35 (2014), pp. 248–254. URL: https://www.latex-project.org/publications/tb111mitt-float.pdf.

[8] I. J. Munezero et al. “Partitioning Microservices: A Domain Engineering Approach”. In: (May 2018), pp. 43–49.

[9] Shmuel Tyszberowicz et al. “Identifying Microservices Using Functional Decomposition”. In: (2018). Ed. by Xinyu Feng, Markus Müller-Olm, and Zijiang Yang, pp. 50–65.

A. Appendix

A.1. First Appendix Section

Figure A.1.: A figure

…