Service Cutter: A Systematic Approach to Service Decomposition

Michael Gysel[1], Lukas Kölbener[1], Wolfgang Giersche[2], and Olaf Zimmermann[1(✉)]

[1] University of Applied Sciences of Eastern Switzerland (HSR FHO), Oberseestrasse 10, 8640 Rapperswil, Switzerland
{michael.gysel, lukas.koelbener}@lifetime.hsr.ch, ozimmerm@hsr.ch

[2] Zühlke Engineering AG, Wiesenstrasse 10a, 8952 Schlieren, Switzerland
wolfgang.giersche@zuehlke.com

Abstract. Decomposing a software system into smaller parts always has been a challenge in software engineering. It is particularly important to split distributed systems into loosely coupled and highly cohesive units. Service-oriented architectures and their microservices deployments tackle many related problems, but remain vague on how to cut a system into discrete, autonomous, network-accessible services. In this paper, we propose a structured, repeatable approach to service decomposition based on 16 coupling criteria distilled from the literature and industry experience. These coupling criteria form the base of Service Cutter, our method and tool framework for service decomposition. In the Service Cutter approach, coupling information is extracted from software engineering artifacts such as domain models and use cases and represented as an undirected, weighted graph to find and score densely connected clusters. The resulting candidate service cuts promise to reduce coupling between and promote high cohesion within services. In our validation activities, which included prototyping, action research and case studies, we successfully decomposed two sample applications with acceptable performance; most (but not all) test scenarios resulted in appropriate service cuts. These results as well as early feedback from members of the target audience in industry and academia suggest that our coupling criteria catalog and tool-supported service decomposition approach have the potential to assist a service architect's design decisions in a viable and practical manner.

Keywords: Functional partitioning · Loose coupling · Knowledge management · Microservices · Service interface design guidelines · Service granularity · Service quality

1 Introduction

In 1972, D. L. Parnas reflected "On the Criteria to Be Used in Decomposing Systems into Modules" [11]. Since then, functional decomposition has remained an important topic in software engineering. As software systems grew and became more complex, software engineers started to distribute modules and procedures over networks, e.g., as

M. Aiello et al. (Eds.): ESOCC 2016, LNCS 9846, pp. 185–200, 2016.
DOI: 10.1007/978-3-319-44482-6_12

remote objects, components or Web services [1]. Architectural styles such as *Service-Oriented Architecture (SOA)* aim at tackling the many design challenges of such distributed systems; however, designing service interface boundaries at the right level of granularity remained an important challenge for SOA practitioners [3, 17]. While partial solutions have been found, two of the related Research Problems (RP) remained open: (RP1) The *architecturally significant requirements* and *stakeholder concerns* to be addressed during service (de-)composition are still not understood fully and have not been documented consistently and comprehensively yet. (RP2) A requirements-driven, repeatable, and scalable *service decomposition method*, to be supported and partially automated by service design tools, has been missing until now.

In this paper, we collect architecturally significant requirements for service decomposition and introduce *Service Cutter*, our knowledge management method and supporting tool framework that assist software architects when they make service design decisions (note that we do not intend to fully automate this decision making process, but rather support it). The remainder of the paper presents our solutions to RP1 and RP2 as well as their validation in the following way: Sect. 2 scopes the context of our work and the research problems solved, and defines our basic service decomposition terminology. Section 3 presents our first research contribution, a coupling criteria catalog for service decomposition; Sect. 4 then defines a novel service decomposition process and an extensible tool architecture that integrates existing graph clustering algorithms to derive candidate service cuts from system specification artifacts. Section 5 presents an implementation of the tool architecture and our validation, which includes action research, two case studies, and performance measurements; Sect. 6 discusses strengths and weaknesses of Service Cutter and presents initial industry feedback. Section 7 concludes and highlights future work.

2 Context, Problem and Supporting Definitions

The impact of service boundary design is far-reaching. Loosely coupled, but highly cohesive services are crucial for the maintainability and scalability of software and allow architects and developers to choose a suitable technology independently for each particular business problem and context. Nevertheless, the decomposition of a monolithic application into services still is not fully understood, even with the rise of *microservices* [16], a contemporary incarnation of SOA principles and patterns combined with modern software engineering practices such as continuous, independent deployment. For instance, a popular introduction to microservices states that "deciding how to partition a system into a set of services is very much an art" [15].

Microservices advocates suggest leveraging *Domain-Driven Design (DDD)* [5] to obtain service boundaries: For instance, instances of the DDD pattern *aggregate* establish composed services that are aligned to consistency constraints, and services derived from *bounded contexts* are aligned to domain model boundaries or team organization structures. Both of these two DDD strategies are suitable approaches to service identification (assuming that one knows how to find aggregates and bounded contexts in the requirements). However, our collective industry experience and a literature review indicate that many more stakeholder concerns have to be taken into

account during service decomposition – in particular, architecturally significant requirements including software quality attributes [2]. We believe that this process can and should be approached in a more structured way. This leads to our first hypothesis:

The driving forces for service decomposition can be presented to architects in a comprehensive and comprehensible coupling criteria catalog.

This criteria catalog, which will be introduced in the next section, assembles 16 decomposition criteria commonly used by architects to frame and guide their architectural decisions. We distilled it in an iterative and incremental way, leveraging consecutive project retrospectives, interviews, and a coupling criteria workshop.

A systematic collection of design knowledge can serve as the foundation for partial automation of analysis and design. This observation leads to our second hypothesis:

Based on the coupling criteria catalog, a system's specification artifacts can be processed in a structured and partially automated way to suggest service decompositions that promote loose coupling between and high cohesion within services.

To investigate whether these two hypotheses hold true, we conceptualized and developed *Service Cutter*, a tool framework architecture and prototype to analyze software engineering artifacts, including use cases and domain models, and to suggest candidate service decompositions.

Service Cutter and its presentation in this paper use the following terminology:

Definitions. The term *service* can be defined both on a logical and on a physical level:

1. A service is the technical authority for a specific business capability [3].
2. A service is accessed remotely through some invocation interface and communication protocol, either synchronously or asynchronously [6].

In order to provide capabilities, a service requires *resources*. We identified three types of resources that serve as the building blocks of services in our approach:

1. Data. A service may have ownership over a subset of a system's data [16]. It then is the only authority allowed to change this data, notifying other services on such changes. The data is often, but not always, stored in a database (then called application state); data exposed at the service interface constitutes its *published language* [5].
2. Operations. A service can encapsulate business rules and calculation (processing) logic. Operations are often, but not always, based on the data owned by the service.
3. Artifacts. An artifact is a snapshot of data or operation results transformed into a specific format. An example is a business report such as monthly sales figures by geography, which was assembled using operations and data.

To facilitate a systematic approach to service decomposition, we generalize these resources with the concept of a *nanoentity* shown in Fig. 1:

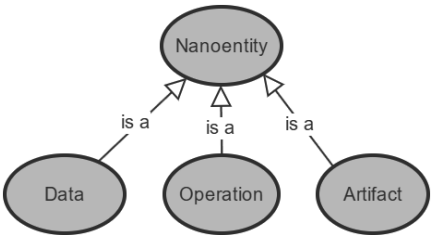

Fig. 1. Data, operations and artifacts generalized into the nanoentity concept.

Service decomposition then can be defined as the process of identifying a set of services and assigning all nanoentities to one (and only one) of these services. A *coupling criterion* represents a particular driving force for service decomposition; such criteria capture architecturally significant requirements and arguments why two nanoentities should or should not be owned and exposed by the same service. *Software System Artifacts (SSAs)* represent the analysis and design artifacts that contain information about coupling criteria; scoring *priorities* weigh the coupling criteria. A *service cut* is the output of a single execution of the service decomposition process.

3 Coupling Criteria Catalog

We conducted a literature review, reflected on past projects, and met for a workshop to assemble our collective, precompiled architecture design experience. We consolidated the results of these knowledge gathering activities in a *coupling criteria catalog* in an iterative and incremental manner. Our coupling criteria catalog aims at serving as a comprehensive, yet not complete collection of architecturally significant requirements and decision drivers for service decomposition. Note that we strived for consensus, clarity, and compactness; hence, not all candidate criteria made it into the catalog. Figure 2 lists the 16 Coupling Criteria (CC) in the final catalog version:

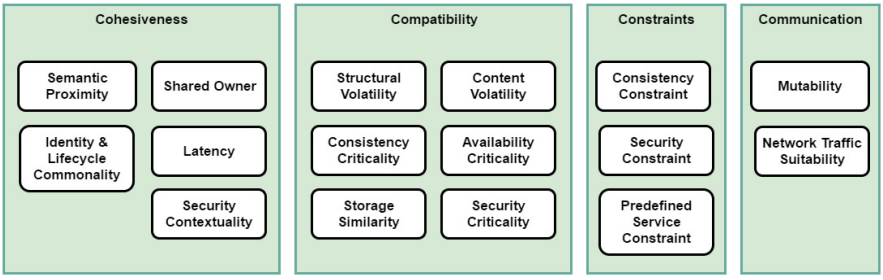

Fig. 2. Coupling Criteria (CC) catalog compiling 16 CC in four categories.

We grouped the CC into four categories in the catalog (to improve readability):

1. Cohesiveness: Criteria describing certain common properties of mutually related nanoentities that justify why these nanoentities should belong to the same service.

An example of a cohesiveness argument is that all nanoentities involved in the realization of a use case should belong to a single service to simplify use case execution.

2. Compatibility: Criteria indicating divergent characteristics of nanoentities. A service should not contain nanoentities with incompatible characteristics. Examples of such characteristics are "high", "eventually", and "weak" for the criterion Consistency Criticality; these data consistency management options are mutually exclusive.

3. Constraints: Criteria specifying high-impact requirements that enforce that certain groups of nanoentities (a) must jointly constitute a dedicated service or (b) must be distributed amongst different services. The fact that a set of nanoentities has to be modified jointly and atomically, e.g. in the same database transaction, forms a strong requirement that justifies to be represented as constraint criterion in the catalog.

4. Communication: Criteria exclusively pertaining to the technical cost of remoting, e.g., *mutability*. Immutable resources do not require complex synchronization means.

All 16 CC are recorded in a common card layout inspired by pattern languages and agile practices. Table 1 introduces this *Coupling Criterion Card (C3)* template:

Table 1. A template for Coupling Criterion Cards (C3).

[Coupling Criteria Identifier and Name]	
Description	[A brief summary of the Coupling Criterion (CC) w.r.t. its impact on/usage of nanoentities]
System Specification Artifacts (SSAs)	[Requirements engineering input and software architecture concepts/deliverables pertaining to this coupling criterion]
Literature	[References to books, articles, and/or blog posts]
Type	Cohesiveness | Compatibility | Constraint | Communication
Characteristics	[Defines a set of possible values for this CC. Only applies to CC of type Compatibility. E.g., "critical", "normal", "low"]

The usage of such C3s makes the catalog structure recognizable and the catalog extensible. Tables 2 and 3 present two examples of filled-out C3 instances.[1]

Eliciting CC instances to reflect the non-functional requirements of a specific software product is a key aspect of analysis and design. Hence, software architects can leverage the CC catalog to establish a common terminology for their design discussions as well as architecture documentation. Moreover, our CC catalog can serve as the basis of a structured, repeatable way to identify, make, and capture related decisions [18]; it serves as *ubiquitous language* [5] for service decomposition.

[1] All 16 coupling criteria cards are published in full length in the Service Cutter wiki on GitHub, https://github.com/ServiceCutter/ServiceCutter/wiki/Coupling-Criteria.

Table 2. The "Identity and Lifecycle Commonality" CC.

CC-1 Identity and Lifecycle Commonality	
Description	Nanoentities that belong to the same identity and therefore share a common lifecycle (create, read, update, delete)
System Specification Artifacts (SSAs)	– Entity-Relationship Models – Domain-Driven Design Entity pattern instances
Literature	Entity definition in Domain-Driven Design [5]: *Some objects are not defined primarily by their attributes* *They represent a thread of identity that runs through time and often across distinct representations*
Type	Cohesiveness

Table 3. The "Semantic Proximity" CC.

CC-2 Semantic Proximity	
Description	Two nanoentities are semantically proximate when they have a semantic connection given by the business domain The strongest indicator for semantic proximity is coherent (joint) access of/to nanoentities within the same use case
System Specification Artifacts (SSAs)	– Coherent access to or updates of nanoentities in use cases (or user stories) – Aggregation or association relationships in an entity-relationship model
Literature	Single Responsibility Principle by Martin [9]: *Gather together the things that change for the same reasons* *Separate those things that change for different reasons* Richardson on microservice decomposition [15]: *There are number of strategies that can help [to partition a system into a set of services]. One approach is to partition services by verb or use case*
Type	Cohesiveness

4 Service Decomposition Concepts and Tool Architecture

To allow architects to leverage the CC catalog and receive service decomposition advice, we created the *Service Cutter* tool framework. Service Cutter derives *candidate service cuts* from user-prioritized coupling criteria (obtained from SSAs) to achieve loose coupling between services and high cohesion within services. To do so, additional design concepts are required, which will be introduced in this section.

Decomposition Input. The input to Service Cutter is a machine-readable representation of selected software engineering artifacts that represent intermediate stages of analysis and design. To represent these artifacts, we introduce *System Specification Artifacts (SSAs).* SSAs serve as data sets from which the Service Cutter can extract the required coupling criteria information. Examples of SSA types are use cases, DDD entities/aggregates, and Entity-Relationship Models (ERMs); e.g., information about

CC-2 Semantic Proximity comes from these two SSA types. We designed additional SSA types to supply information that is not contained in existing ones (e.g., shared owner groups, predefined services, separated security zones and security access groups). The Service Cutter wiki provides detailed explanations and a reference of these nine types of SSAs (called "user representations" in the prototype).[2]

Figure 3 specifies the dependencies of coupling criteria and SSAs. For instance, information about CC-16, Security Constraint, can be obtained from the SSA "separated security zones". Security zones group nanoentities by their diverging privacy requirements, e.g. sensible personal information vs. unclassified, public data.

Decomposition Process. Figure 4 specifies the service cutting process in BPMN.

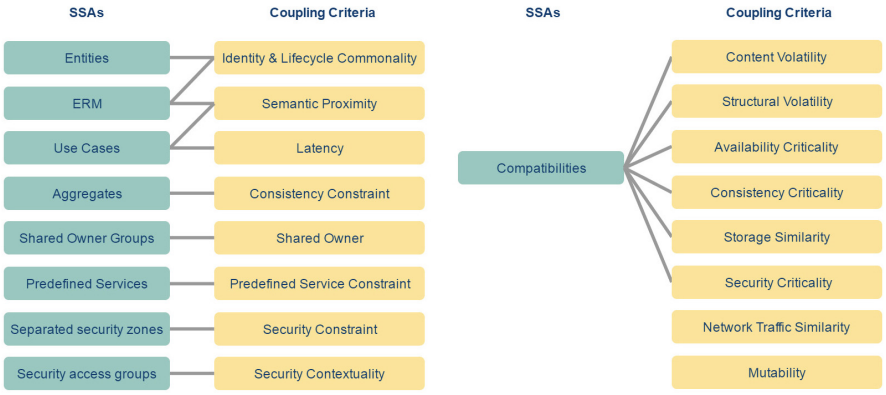

Fig. 3. Dependencies between System Specification Artifacts (SSAs) and CC.

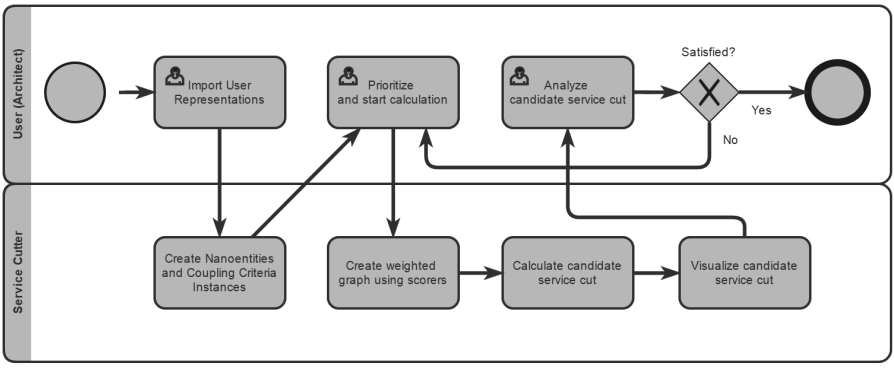

Fig. 4. Serving decomposition process (human vs. automated/tool-supported tasks).

[2] https://github.com/ServiceCutter/ServiceCutter/wiki/User-Representations.

Service Cutter processes the provided SSA instances and extracts nanoentities as well as coupling criteria instances from them. *Prioritized* coupling criteria and SSAs are transformed into an undirected, weighted graph; nodes represent nanoentities, and the weights of edges indicate how cohesive and/or coupled two nanoentities are.

Algorithm Integration. We then employ *clustering algorithms* on this graph to find candidate service cuts. Our concepts and tool architecture are designed to be general enough to allow the inclusion of multiple algorithms; e.g., a programming interface is provided which can be implemented for any clustering algorithm that is based on undirected, weighted graphs. At present, we included Java implementations of two algorithms, namely Girvan-Newman [10] and the Epidemic Label Propagation (ELP), originally defined by Raghavan and later refined by Leung et al. [14]. A comparison of and rationale for the selection of these two different approaches can be found in [7]. For instance, the two algorithms differ from each other in their (non-)deterministic behavior; only one of them required a number-of-clusters in parameter.

Results of a deterministic algorithm like Girvan-Newman can be reproduced by running the algorithm repeatedly using the same input data. The impact of different input data, scoring values and priorities can therefore be analyzed as the algorithm itself does not include a random element. A non-deterministic algorithm like ELP (Leung) complicates analysis, as changes in the results do not always result from input changes. Furthermore, results always need to be safely persisted and reloaded since they cannot be reproduced reliably. An element of randomness is not necessarily a disadvantage: Running multiple algorithm cycles presents different solutions and outlines where the difficult architectural decisions reside.

Providing the number of clusters as a parameter to the algorithm has the advantage of analyzing the service decomposition with any possible number of services. This feature can be used to better understand the structure and coupling between parts of the system when running the algorithm with varying input. Requesting a high number of services, for instance, may indicate how services can be decomposed further; a small predefined service number allows systems to gradually emerge from a monolithic architecture to service orientation. However, algorithms requiring the number of services as input shift the responsibility to answer this critical question back to the user; as architects are often prejudiced on the number of services their system should be composed of, this is not always desirable. Letting Service Cutter suggest not only the content of each service, but also the number of services (as ELP does) challenges the user to reassess his/her ideas against the suggested candidate service cuts.

Priority Scoring. The analysis and processing of coupling criteria uses a weighted graph and *scorers*. The weight on an edge between two nanoentities is the sum of all scores per CC multiplied by their priorities. Table 4 illustrates the calculation:

Table 4. An exemplary calculation of the weight of an edge.

Coupling criterion	Score	Priority	Result
CC-1: Semantic Proximity	4	1	$4 * 1 = 4$
CC-7: Availability Criticality	2.5	5	$2.5 * 5 = 12.5$
CC-9: Consistency Constraint	8	3	$8 * 3 = 24$
Total weight			$4 + 12.5 + 24 = 40.5$

The *score* is a number from −10 to +10. A score of +10 expresses that these two nanoentities should definitely reside in the same service according this coupling criterion. A score of −10 therefore represents the opposite extreme, i.e., that the nanoentities should be placed into different services.

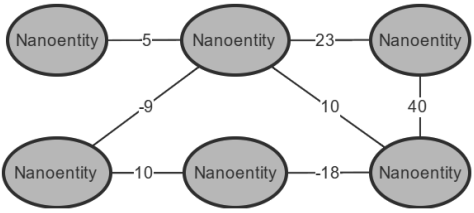

Fig. 5. Weighted edges representing the coupling connect the nanoentities.

The calculation is performed for every link between nodes with coupling information; Fig. 5 shows an example. The calculation depends on the involved coupling criteria; the scorers map coupling criteria to actual numbers used to construct the weighted graph. Table 5 maps CCs to the five types of scorers that differ in their calculation logic:

Table 5. Coupling criteria and the scorers calculating the weight of the edges.

Coupling criterion	Scorer type
Identity and Lifecycle Commonality *Shared Owner* *Latency* *Security Contextuality* *Consistency Constraint*	*Cohesive Group Scorer* Nanoentities in a cohesive group should remain together in one service. All relations between nanoentities in a group are scored +10
Semantic Proximity	*Semantic Proximity Scorer* The joint access to a pair of nanoentities is counted and mapped to an even distribution between 0 and 10
Structural Volatility *Consistency Criticality* *Storage Similarity* *Content Volatility* *Availability Criticality* *Security Criticality*	*Characteristics Scorer* To achieve homogenous services, this scorer sets a penalty of −1 to −10 to relations with diverging requirements
Security Constraint	*Separated Group Scorer* Sets a score of −10 to all nanoentities that belong to a group other than the current one
Predefined Service Constraint	*Exclusive Group Scorer* Same as Cohesive Group, but also adds a penalty of −10 to nanoentities not in the group
Mutability *Network Traffic* *Suitability*	Not defined and implemented yet

A detailed description of the scorers in Service Cutter can be found in [7].

5 Evaluation via Prototyping, Case Studies, Action Research

We validated our research results via implementation, case study, and action research. Service Cutter's current implementation supports a basic feature set that realizes the structured approach of splitting a system into discrete, loosely coupled services:

- 14 out of 16 coupling criteria from Sect. 3 are implemented (see Table 5).
- All nine System Specification Artifacts (SSAs) that represent user input (see Fig. 3 in Sect. 4) can be imported in the form of custom JSON files.
- Seven criteria priorities, in the prototype casually defined as "T-Shirt sizes" (IGNORE, XS, S, M, L, XL, XXL) allow users to characterize the context of a system by valuating the coupling criteria in relation to each other.
- The suggested candidate service cuts and their dependencies are visualized.
- The published language [5] of a service pair (including the data transferred to and from the invoked service) is exposed via the involved nanoentities.

Figure 6 features a candidate service cut for the "cargo tracking" domain model from [5]. This candidate service cut consists of three services A, B and C (larger squares), each owning a set of (cohesive) nanoentities represented as small squares:

Arrows between two services (e.g., Service A and Service B) indicate a dependency between them. The resulting published language, which characterizes the amount of coupling between these services in terms of the shared understanding about the nanoentities that are exposed at the service boundary, is also shown.

Release 1.1 of the Service Cutter implementation is available on GitHub[3]. This prototype consists of two components implemented in Java and JavaScript (using Spring Boot, Spring MVC, AngularJS, and JHipster), RESTful HTTP Web services wrapping the scoring logic, and a Web application for input and output visualization.

Validation Approach and Results. To further validate the implemented concepts, we assessed the candidate service cuts of the following two case studies:

1. A fictitious "Trading System" for which we forward-engineered the requirements, drawing on industry experience with financial services software.
2. The DDD sample application "Cargo Tracking" that accompanies the DDD book [5]; we reverse engineered the requirements for this scenario from the existing implementation that is available on SourceForge.[4]

To objectify the validation and have a comparison baseline, we defined expected service cuts for both systems according to our experience in service design; to reduce bias, we developed a service design checklist for this task.[5] Next, we defined three result categories in order to rate the candidate service cuts:

[3] https://github.com/ServiceCutter/ServiceCutter.

[4] https://sourceforge.net/projects/dddsample/.

[5] https://github.com/ServiceCutter/ServiceCutter/wiki/Decomposition-Questionnaire.

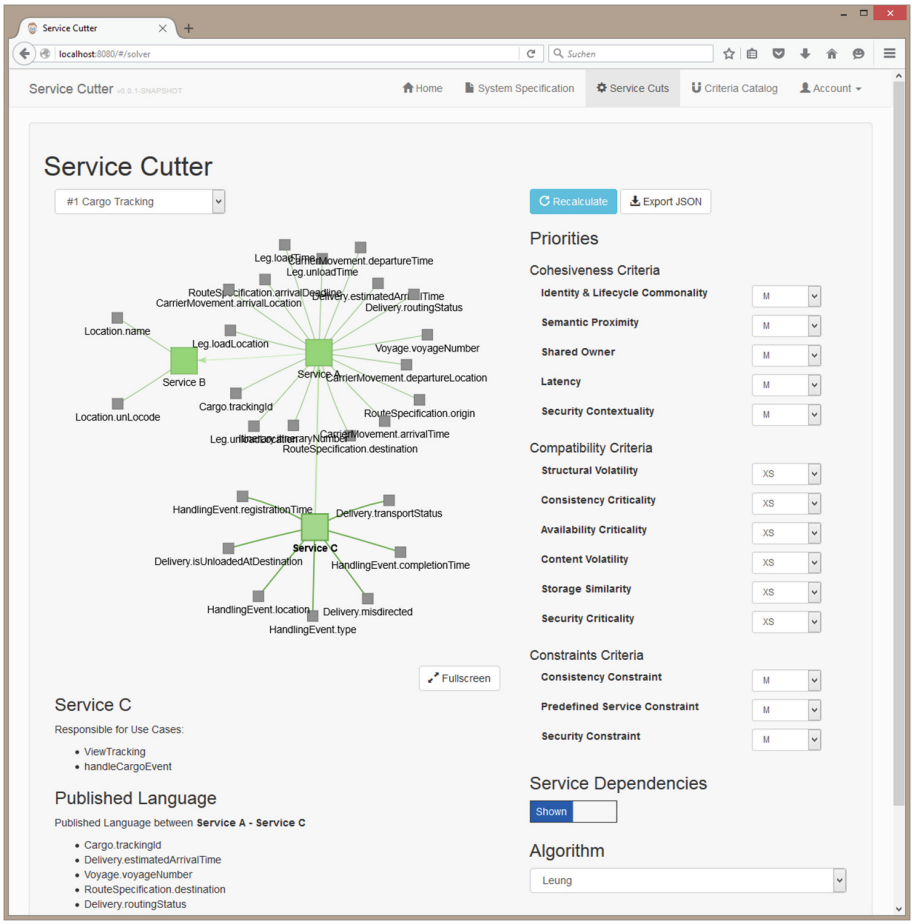

Fig. 6. Screenshot of Service Cutter presenting a candidate service cut.

A: Excellent service cut. The cut (i.e., suggested service decomposition) does not follow the way we expected, but we find reasons why the cut makes sense from an architect's perspective. It therefore improves our own view of the analysed system.
B: Expected service cut. The cut meets and therefore validates our expectations.
C: Unreasonable service cut. There is a mismatch between the cut and the expected one, and we do not find any reasons why this cut would be beneficial.

To be able to assess the quality of the output of Service Cutter, we use a four-level classification: An *excellent* output contains zero unreasonable service cuts and at least one excellent service cut (i.e., a cut in category A). A *good* output contains zero unreasonable service cuts (C). An *acceptable* output contains at most one unreasonable service cut (C). A *bad* output contains two or more unreasonable service cuts (C).

Table 6 summarizes the decomposition results for both systems. Both algorithms, Girvan-Newman and ELP (Leung), were able to produce acceptable or good service cuts (but not in all cases):

Table 6. Assessment of service cuts for analyzed systems (case studies).

Evaluated Application	Girvan-Newman	ELP (Leung)
Trading System	Good output	Good (note: in some exceptional cases, Leung produced acceptable *and* excellent output)
Cargo Tracking System	Bad output	Acceptable

Both test systems contain approximately 20 nanoentities. To analyze Service Cutter's performance behavior with more complex systems, we conducted additional performance tests. These tests are derived from the trading system; all nanoentities and SSAs were replicated and scaled up 60 times to create larger and more complex domain models and graphs. These load tests measure the runtime for graph creation and clustering algorithm and leave out data import and visualization. The tests were conducted on a Windows 10 developer notebook with an Intel i5 2.2 GHz CPU and 8 GB RAM as documented in detail online.[6] Figure 7 shows the test results.

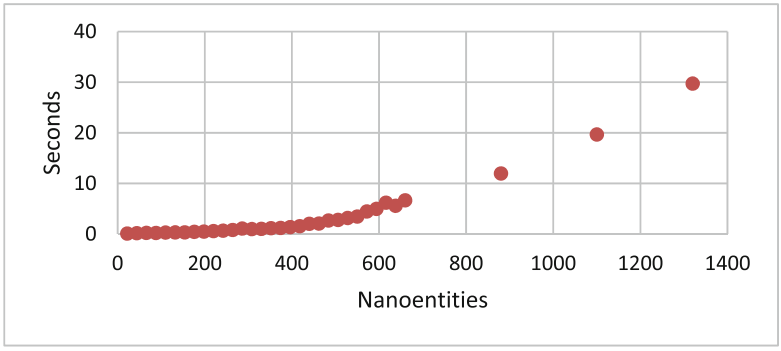

Fig. 7. Performance test results: service cut calculation (scaled up sample application)

The calculation for systems with up to 600 nanoentities is done in less than five seconds, which we consider reasonable. Around 75 % of the time used is consumed by graph creation whereas the clustering algorithm only uses around 25 % of the time. Hence, our Java code building the graph based on the imported data could be analyzed and improved to improve runtime performance even further.

[6] https://github.com/ServiceCutter/ServiceCutter/wiki/Runtime-Performance-Tests.

6 Discussion: User Feedback, Pros and Cons, Related Work

User Feedback. We presented the Service Cutter concepts and their implementation to more than 20 members of the target audience (i.e., software engineers and architects with experience in designing SOAs), and one of the authors of the paper applied Service Cutter to a single project case (as a form of technical action research). The systematic overall approach was appreciated and considered to be promising; it was pointed out that Service Cutter cannot only be used in an SOA context, but also be used to split modules without remote interfaces (with adjusted CC priorities).

The template-based coupling criteria cards were generally appreciated, but some of the current texts were assessed to be too terse (by one provider of feedback); a more elaborate, but not yet verbose wording was requested. The naming of some coupling criteria in our catalog also was challenged. An example is "CC-13 Network Traffic Suitability", which covers the more common and basic concept of *throughput* (which in turn is one facet of the top-level quality attribute *performance*). Furthermore, system and process assurance *audit compliance* [8] was suggested to be added as a compatibility criterion; further research is required to investigate how to integrate such a composite and complex, possibly even recursive criterion into Service Cutter.

Finally, our selection of two clustering algorithms was questioned, and it was suggested to only integrate deterministic algorithms that do not require the number of clusters as a parameter. This critique pertains to the current tool implementation only; the Service Cutter concepts from Sects. 3 and 4 do not rely on any particular algorithm. Due to the generality of our concepts and the modular, extensible architecture of their implementation, we expect the effort to integrate other algorithms into the Service Cutter framework to be in the range of a few person days per algorithm.

According to the feedback of our industry project partner, who leads an architect and developer community in professional services, Service Cutter and its underlying reasoning represent a sound framework to prepare and back architectural decisions. More specifically, it allows architects to study the impact of weight variations on the resulting candidate service cuts. Questions like "what, if security wasn't an issue here" can be answered easily by changing the respective scoring priority of criterion "security criticality". When used with care, Service Cutter can improve the credibility of architects involved in critical architecture assessments (evaluations) significantly. The SSAs and coupling criteria can also be used to educate junior architects or students on the driving forces of service decomposition.

Benefits. From our internal and external validation activities, we can conclude that Service Cutter offers a number of advantages to service architects: The coupling criteria catalog indeed collects relevant architecturally significant requirements and decision drivers for service decomposition, and it does so in an accessible, reusable, and extensible way. It therefore contributes to the body of reusable architectural decision knowledge as envisioned in our previous work [18].

Service Cutter suggests candidate service cuts that are obtained from commonly used analysis and design artifacts, such as use cases and domain models, via a nanoentity abstraction and the coupling criteria. By expecting several such analysis and design artifacts, Service Cutter challenges its users (i.e., service architects) to reflect

which stakeholder input and non-functional quality characteristics are relevant for his/her system (and architecture design process). Hence, service architects might use these artifacts as a checklist and stimulus for the requirement engineering.

The candidate service cuts verify and/or challenge the architect's expectations regarding the number of services and their interface definitions. Both green field scenarios and iterative approaches for migrating a monolith to services are supported.[7]

Drawbacks and Liabilities. The benefits that we could observe during our evaluation activities come at a price; usage of Service Cutter concepts and their implementation during these activities has unveiled some (expected) drawbacks and liabilities.

Significant effort is required to enter SSAs (such as use cases and domain models) in JSON; in future versions, we plan to import them, e.g., from UML modeling tools.

We are aware of the risk of a "pseudo accuracy" effect. It is subject to debate whether service design work, dealing with rather diverse requirements (some of which are hard to quantify) can really be delegated to algorithms that look for an aggregated optimal solution. Architects traditionally apply their tacit knowledge and "gut feel" when making the relate decisions; they are biased. This discussion can be seen as the SOA variant of the more general discussion on "a rational design process: how and why to fake it" [12]. However, we believe our approach to be valuable even when being confronted with a healthy amount of skepticism – relevant design questions are asked and related criteria listed, and the relation between these concerns and the user input in SSAs is unveiled. Furthermore, a checklist effect occurs; discussions among collaborating architects are stimulated.

Other drawbacks and liabilities concern framework architecture design and extensibility. First and foremost, the clustering algorithms that are currently integrated possibly should be complemented with additional ones due to the only partially satisfying evaluation results. Algorithmic complexity is a major source of performance limitations and therefore has to be taken into account in any such future algorithm selection decisions; fortunately, clustering algorithms with linear complexity exist.

As the Service Cutter framework continues to evolve, additional validation and evaluation activities work will be required. For instance, it has to be verified that the tool performance does not degrade significantly when processing even larger amounts of user input that go beyond scaled up sample data and case studies (e.g., complex domain models from enterprise information systems).

Related Work. Quality attribute-driven design has been an important research topic in the software architecture community for many years [2, 11]; the specific requirements and constraints of service-oriented architectures and microservices have also been investigated and related methods proposed [4, 13, 17]. Such methods are complementary to the approach presented in this paper, providing an overall frame for the use of Service Cutter, as well as input for coupling criteria, SSAs, and priority scores.

Other research areas in service-oriented computing include service discovery and runtime topology lookup (e.g., in clouds), dynamic service matchmaking, service composition into business processes and workflows, quality-of-service awareness,

[7] Explained on GitHub: https://github.com/ServiceCutter/ServiceCutter/wiki/Usage-Scenarios.

policies, and agreement, as well as service management. These efforts have different goals than Service Cutter, which aims at assisting architects making design decisions; however, well-crafted service cuts can be seen as a prerequisite for the successful application of any advanced service-oriented computing concepts and technologies. In our future work, we therefore consider to include additional criteria and SSAs that represent the concepts from these research efforts as they mature.

7 Summary and Outlook

In this paper, we presented Service Cutter, a systematic approach to system decomposition, which has been a relevant problem since the very origins of program modularization and software engineering. Service Cutter advances the state of the art (a) with the concept of coupling criteria cards, (b) 16 instances of such cards (harvested from practical experience and the literature), and (c) an extensible service decomposition tool framework architecture that integrates graph clustering algorithms and features priority scoring starting from nanoentities and nine types of analysis and design specifications (including domain models and use cases). This structured and extensible combination of a criteria-driven method with supporting architectural knowledge and a design optimization and visualization tool paves the way towards the desired engineering approach to service interface and service granularity design.

We evaluated Service Cutter via implementation (integrating two existing graph clustering algorithms), a combination of action research and case study investigations, and load tests. The validation results and additional user feedback indicate that the proposed semi-automated approach to service decomposition works as designed and has the potential to benefit practitioners significantly. While the suggested service cuts did not always meet all early adopters' expectations, artifact input and coupling criteria were regarded adequate; the proposed decomposition process was appreciated.

While our early experiences with the presented structured, partially automated (i.e., tool supported) approach are promising, work remains to be done both on the conceptual (research) level, as well as on the implementation (engineering) level. For instance, further enhancements of Service Cutter may include seamless integrations of the analysis and design tool chain members so that SSAs can be extracted from other tools automatically. We discussed other directions for future work in Sect. 6; related development issues are tracked in the open source release of Service Cutter.

References

1. Alonso, G., Casati, F., Kuno, H.A., Machiraju, V.: Web Services – Concepts, Architectures and Applications. Data-Centric Systems and Applications. Springer, Heidelberg (2004)
2. Cervantes, H., Velasco, P., Kazman, R.: A principled way of using frameworks in architectural design. IEEE Softw. **30**(2), 46–53 (2013)
3. Dahan, U.: The Known Unknowns of SOA, Blog Post, November 2010. http://udidahan.com/2010/11/15/the-known-unknowns-of-soa/

4. Erradi, A., Anand, S., Kulkarni, N.: SOAF: an architectural framework for service definition and realization. In: Proceedings of SCC 2006. IEEE Computer Society (2006)
5. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Pearson Education, Upper Saddle River (2003)
6. Fowler, M.: Inversion of Control Containers and the Dependency Injection Pattern, Online Article, January 2014. http://www.martinfowler.com/articles/injection.html
7. Gysel, M., Kölbener, L.: Service cutter – a structured way to service decomposition. Bachelor thesis, HSR Hochschule für Technik Rapperswil (2015). https://eprints.hsr.ch/476/
8. Julisch, K., Suter, C., Woitalla, T., Zimmermann, O.: Compliance by design – bridging the chasm between auditors and IT architects. Comput. Secur. **30**(6–7), 410–426 (2011). Elsevier
9. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River (2003)
10. Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E **69** (2004). arXiv:cond-mat/0308217
11. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
12. Parnas, D.L., Clements, P.C.: A rational design process: how and why to fake it. IEEE Trans. Softw. Eng. **12**(2), 251–257 (1986)
13. Papazoglou, M., van den Heuvel, W.J.: Service-oriented design and development methodology. Int. J. Web Eng. Technol. (IJWET) **2**(4), 412–442 (2006). Inderscience Enterprises
14. Raghavan, U.N., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale network. Phys. Rev. E **76** (2007). arXiv:0709.2938
15. Richardson, C.: Microservices: Decomposing Applications for Deployability and Scalability. InfoQ article, May 2014. http://www.infoq.com/articles/microservices-intro
16. Zimmermann, O.: Microservices tenets: agile approach to service development and deployment. Overview and vision paper, SummerSoC 2016. J. Comput. Sci. Res. Dev. (CSRD), Springer (to appear)
17. Zimmermann, O., Krogdahl, P., Gee, C.: Elements of Service-Oriented Analysis and Design. IBM developerWorks, July 2004
18. Zimmermann, O., Wegmann, L., Koziolek, H., Goldschmidt, T.: Architectural decision guidance across projects. In: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 85–92. IEEE Computer Society (2015)