View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Brunel University Research Archive

Microservice Ambients: An Architectural Meta-modelling Approach for Microservice Granularity

Sara Hassan
School of Computer Science
University of Birmingham
Birmingham, UK
ssh195@cs.bham.ac.uk

Nour Ali
School of Computing, Engineering and Mathematics
University of Brighton
Brighton, UK
N.Ali2@brighton.ac.uk

Rami Bahsoon
School of Computer Science
University of Birmingham
Birmingham, UK
r.bahsoon@cs.bham.ac.uk

***Abstract*—Isolating fine-grained business functionalities by boundaries into entities called microservices is a core activity underlying *microservitization*. We define microservitization as the paradigm shift towards microservices. Determining the optimal microservice boundaries (i.e. microservice granularity) is among the key microservitization design decisions that influence the Quality of Service (QoS) of the microservice application at runtime. In this paper, we provide an architecture-centric approach to model this decision problem. We build on ambients — a modelling approach that can explicitly capture functional boundaries and their adaptation. We extend the aspect-oriented architectural meta-modelling approach of ambients — AMBIENT-PRISMA — with *microservice ambients*. A microservice ambient is a modelling concept that treats microservice boundaries as an adaptable first-class entity. We use a hypothetical online movie subscription-based system to capture a microservitization scenario using our aspect-oriented modelling approach. The results show the ability of microservice ambients to express the functional boundary of a microservice, the concerns of each boundary, the relationships across boundaries and the adaptations of these boundaries. Additionally, we evaluate the expressiveness and effectiveness of microservice ambients using criteria from Architecture Description Language (ADL) classification frameworks since microservice ambients essentially support architecture description for microservices. The evaluation focuses on the fundamental modelling constructs of microservice ambients and how they support microservitization properties such as utility-driven design, tool heterogeneity and decentralised governance. The evaluation highlights how microservice ambients support analysis, evolution and mobility/location awareness which are significant to quality-driven microservice granularity adaptation. The evaluation is general and irrespective of the particular application domain and the business competencies in that domain.**

***Keywords*-microservices; meta-modelling; granularity; ambients**

I. Introduction

Microservitization is a shift towards transforming services/components into microservices — more fine-grained and autonomic services that isolate fine-grained business functionalities by boundaries and interact through standardised interfaces [1]. Microservitization is rapidly increasing; many distributed and cloud-based systems have evolved from monolithic to microservices architectures. Examples of large-scale applications which adopt microservices include Netflix [2], Amazon [3] and Uber [4]. Netflix for example receives around one billion streaming requests everyday [5], thereafter routing each request through an API to multiple back-end microservices [2]. Each microservice encapsulates a fine-grained business functionality.

Isolating business functionalities aims at enhancing the autonomy and replaceability of the individual microservice(s) [1]. This in turn can enhance decentralised governance of the microservices, where each microservice encapsulates fine-grained business functionality. Microservitization hopes to enhance the flexibility of large scale distributed applications to make them better cope with operation, maintenance and evolution uncertainties. Such flexibility promises improvement to the maintenance costs and quality of service (QoS) provision to system users.

With the hype and increased interest of software industries in microservices, there is still a general lack of systematic approaches that model microservices design decisions, including deciding the optimal microservice boundaries — optimal microservice granularity level. Bridging this gap is a prerequisite for advancing the adoption of this paradigm and for facilitating design and runtime analysis that support its operation, maintenance and evolution. In [1] we formulated this design decision as a runtime decision problem and we set a roadmap for a self-adaptive solution to it. “Splitting too soon can make things very difficult to reason about. It will likely happen that you (the software architect) will learn in the process. [6].”

The runtime context is therefore more suitable to this problem since much of the uncertainties that relate to the choice of the optimal level of granularity and the expected behaviour of the system can not be fully captured at design time [1]. *In this paper, we call for a systematic, flexible and expressive architectural modelling and analysis support for representing microservices and managing their granularities to model this decision problem.*A systematic architecture-centric approach for modelling microservice granularity provides the appropriate level of abstraction for managing this

runtime decision problem. In particular, this approach has the promise to scale the analysis of this problem. Reasoning about microservice granularity at the architectural rather than code level can facilitate analysis of systems that exhibit heterogeneity and decentralised governance — as is the case with microservice applications. The heterogeneity of tools supporting microservice architectures calls for flexibility in modelling the granularity decision problem in a technology independent way. Architectural modelling of microservices in particular is a pressing issue due to the lack of standardisation for this young research field [7]. Although intuitively the main trigger for microservitization is "people finding they have a monolith that's too big to modify and deploy [8]", we have not encountered any architectural modelling support in the literature to manage the adaptation from the "too big" or "too small" services to the "good enough" level of granularity.

The novel contribution of this paper is an architecture-centric modelling concept for microservices. The concept extends ambients [9], [10] (explained in Section II) by introducing *microservice ambients*. Microservice ambients provide the primitives for modelling microservices and and treat microservice boundaries as adaptable first-class entities. Microservice ambients use "aspects" to define the adaptation behaviour needed to support changes in granularity at runtime. Aspects flexibly separate cross-cutting concerns (and thereby scope) of each boundary. We introduce the *granularity adaptation aspect* to help define the runtime triggers for granularity adaptation to be used as the mciroservice(s) are monitored at runtime. When any of these triggers is invoked, the graphical notation of microservice ambients provides architectural modelling support to express the candidate solution for granularity adaptation. The transition (by merging or decomposing a microservice ambient(s)) from an initial architecture to a chosen candidate is captured in the granularity adaptation aspect of the microservice ambient.

We use a hypothetical online movie subscription-based system (Section III) to demonstrate the flexibility and expressiveness of the modelling approach in capturing microservitization scenarios (Section VI). We evaluate our modelling approach using properties from ADL classification frameworks [11]–[13] since microservice ambients essentially support architecture description for microservices. The evaluation focuses on the fundamental modelling constructs of microservice ambients and how they support microservitization properties such as utility-driven design, tool heterogeneity and decentralised governance. The evaluation also highlights how microservice ambients support runtime analysis, mobility and location awareness; all of which are significant to quality-driven microservice granularity adaptation. The evaluation is general and irrespective of the particular application domain and the business competencies in that domain.

II. Background

Ambient-PRISMA [14] extends traditional architectural elements with a new kind of element called an ambient inspired from Ambient Calculus [9]. Ambients are architectural elements that "coordinate a boundary, model the notion of location and provide mobility support to other architectural elements [10]." Ambients locate other architectural elements in their boundary and manage them. Therefore, ambients can be utilised as runtime modelling analysis tools, where the parent ambient manages the interaction between its children and exterior architectural elements. For example, in Figure 6.a the *MovieReview* ambient is the parent of *RevPolM* and *RevInfoReqM* microservice ambients, so any access to these children ambients from outside the parent ambient is managed by it.

AMBIENT-PRISMA separates the behaviour of cross-cutting concerns through a set of aspects. Aspects give a white-box view of each ambient's boundary. Ambient-PRISMA is supported by a specification language that allows an aspect-oriented description of the software architecture. Crucially, this specification language provides just enough insight into the behaviour (i.e. concerns) of each ambient to model an architecture expressively — capturing the behaviour within ambient and the relationships across ambients.The aspects communicate through weaving relationships. Each aspect fulfils its concern by utilising interface services. Many aspects can utilise the same interface service if need be. Ambients publish interface services through ports. Attachments represent the communication channel between a port of an ambient and any architectural element located inside or outside that ambient. The interface services that we utilise for the granularity problem are [10, p8,p9]:

- *newAmbient* creates a new ambient by providing the name of the ambient as a parameter.
- *addChild* adds a new architectural element in the boundary of an ambient.
- *removeChild* removes an architectural element that is located in the boundary of an ambient.

III. Motivating Example: Microservice Boundaries

We use a hypothetical online movie streaming subscription-based system to demonstrate the significance of systematic, flexible, expressive architectural modelling and analysis support to the microservice granularity problem.

System users are allowed to view movies on multiple browsers upon providing their personal information and setting up a subscription scheme. Personal information about system users and about subscription sales are stored in a local database. Displayed media content is regulated by age restriction policies. The system also allows its users to view and upload movie reviews. Reviews can be numeric ratings and/or written reviews. Written reviews are regulated

according to policies defined by the firm before they are made accessible to other system users. The system architects and developers are currently based in a different department (or "silo") from the regulators that define user review and movie content policies. This brief specification of the system is inspired by the Promise requirement repository [15].

For illustration we assume that the architects adopt a microservitization approach when building this system. They henceforth need to address the microservice granularity challenge. To address this challenge, we assume the architects need to construct a solution space of architectures with varying granularity levels. We assume the intuition of the architects when constructing these candidates is to utilise domain-driven design concepts, such that each bounded context represents an independent area of the domain (inspired by [16], [17]). Independence here implies that this domain has consistent rules " in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas [17]" and these rules only apply within that bounded context. Each modular boundary represents a single microservice within a bounded context. The intuition here is that a modular boundary encapsulates more concretely the related functionalities that belong to the same bounded context.

Furthermore, we assume the architects define decomposition rules such as XOR and OR to represent the relationships between the respective modular boundaries. When presented to stakeholders, these intuitions can raise the following challenges (Cs) (among others):

- **C1**: What is the difference between a "modular boundary" and a "bounded context"?
- **C2**: How does the interaction between modular boundaries differ for XOR and OR decompositions?

IV. Problem Definition

Reflecting on Section III, we describe the requirements of a systematic architecture-centric approach to model microservice granularity:

- **Requirement 1:** The approach shall explicitly capture the primitives for granularity adaptation — microservice boundaries — as first-class entities. Boundaries are primitives for granularity adaptation because they are the effectors/actuators of granularity adaptation design decisions.
- **Requirement 2:** The approach shall promote flexibility and expressiveness in the modelling microservice granularity behaviour, thereby supporting runtime analysis of this behaviour. Crucially this analysis needs to optimise for autonomy of computation and independent deployability of the microservices. The objective of the analysis is to avoid aggressive decomposition of functionality.

The concept of ambients is particularly attractive for the requirements above. *We extend ambients by introducing a microservice ambient type. A microservice ambient as an architectural element allows modelling the boundary of computation of a microservice (Requirement 1, addressing C1).* Aspects, weaving relationships, ports and attachments model the impacts on the concerns and relationships when these boundaries are adapted. *We enrich the microservice ambient with a granularity adaptation aspect to express microservitization scenarios as a set of distinct conditions and decomposition/merging steps. This in turn facilitates runtime analysis for different quality trade-offs for different granularity levels of a microservice ambient (Requirement 2, addressing C2)).* Capturing granularity adaptation behaviour as a transactional set of decomposition/merging steps to facilitate runtime analysis is a novel contribution of this work. Other ADLs we have examined in the literature either assume static modelling of the architectural elements or provide little support for their evolution (e.g. by inheritance or component replication); we discuss this further in Section VI.

V. Microservice Ambient Description

A microservice ambient uses meta-services and meta-properties to reflect on its behaviour and invoke meta-services at runtime. These meta-properties and meta-services are defined in the meta-model [10]. A microservice ambient can encapsulate further microservice ambients in a hierarchical manner. As long as the autonomy of computation (and thereafter independent deployability) are enforced by the boundary, it is fair to assume each microservice ambient in a hierarchy will be instantiated as a concrete microservice. Nevertheless, each microservice will be of a different granularity level, depending on the hierarchy position of the microservice ambient instantiated.

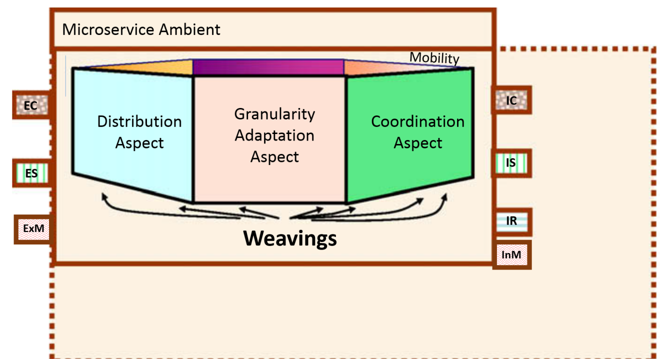


Figure 1. White-box view of a microservice ambient

A microservice ambient can coordinate with its parent ambient (or microservice ambient) to fulfil the behaviour defined by its aspects. Additionally, the microservice ambient can act as an autonomic element that employs the phases of the MAPE-K loop to model the runtime granularity adaptation [18]. The MAPE-K loop defines the stages for

an autonomous element to plan the adaptation of a running system: monitoring the running system, analysing monitored data, planning the adaptation actions, executing the plan on the running system, and finally updating the knowledge base of the autonomous element for future planning optimisation. However, in this paper, we focus more on providing architectural modelling support for the decision problem rather than stating a decision-making method for choosing a granularity adaptation candidate.

A microservice ambient must have at least 4 aspects to support architectural modelling for granularity: a granularity adaptation aspect (our novel contribution), a mobility aspect (inherited from ambients [10]), a coordination aspect (inherited from ambients [10]) and a distribution aspect (inherited from ambients [10]). We elaborate on the role of each aspect below. The overall white-box view of a microservice ambient is shown in Figure 1.

- **Granularity adaptation aspect:** This aspect is responsible for describing the behaviour of how a microservice ambient adapts its granularity. The aspect utilises the *newAmbient,addChild,removeChild,addAttachment and removeAttachment* interface services inherited from the definition of ambients (Section II). At runtime, the granularity adaptation aspect monitors parameters (e.g., change rate, failure rate) indicating QoS. It then utilises the inherited interface services to trigger granularity adaptations in response to changes in the runtime parameters.
- **Mobility aspect:** In the context of supporting granularity adaptation, the mobility aspect allows microservice ambients to enter or exit the boundaries of a parent microservice ambient.
- **Coordination aspect:** At a high level, the coordination aspect redirects calls from external architectural elements to internal architectural elements inside an ambient. It also manages the redirection of calls from internal architectural elements to external ones.
- **Distribution aspect:** The role of the distribution aspect is to make an ambient aware of its hierarchical position by storing the name of its parent ambient. The predefined weaving relationship between the mobility and distribution aspect ensures the distribution aspect manages this awareness when adaptations in the hierarchy are triggered by the mobility aspect.

Microservice ambients inherit all the invariants imposed on ambients [10]. The introduction of the granularity adaptation aspect however introduces an extra invariant specific to microservice ambients. All microservice ambients must implement a *granularity adaptation* aspect whose concern is granularity (***invGranAdapt***). In addition, the following weaving and port constraints are introduced for the microservice ambient:

- The *GranAdapt* aspect definition implement an *IMonitor* interface to define the runtime monitoring requirements of the granularity adaptation aspect. Publishing and requesting the services of this interface indicates the need for 2 additional ports in a microservice ambient:
  - An *InMonitorPort* is required to publish/request the runtime monitoring interface service to architectural elements inside the microservice ambient.
  - An *ExMonitorPort* is required to publish/request the monitoring service to architectural elements outside the microservice ambient. The *ExMonitorPort* of children microservice ambients is connected to the *InMonitorPort* of its parent microservice ambients.
- The granularity adaptation aspect introduces a weaving relationships between itself and the distribution aspect.

```
Granularity Aspect GranAdapt using IMonitor

Attributes
Sources: Set (Source);
SourceNum:Integer
....
Transactions in DECOMPOSE (input AESet: Set(ArchitecturalElement)):
        New1 = newAmbient(AmbName: string);
        New2 = newAmbient(AmbName: string);
        New1.addChild(AESet.nth element.getChildren().subset1);
        New2.addChild(AESet.nth element.getChildren().subset2);
        newSource ("new1",new1,ParameterList);
        sources.add(new1);
        newSource ("new2",new1,ParameterList);
        sources.add(new2);
        sources.delete(AESet. nth element);
        recordReading("new1",ParamName,ParameterKind)
        recordReading("new2",ParamName,ParameterKind)

Transactions in MERGE (input AESet: Set(ArchitecturalElement)):
        ChildrenA  = AESet.nth element.getChildren();
        ChildrenB = AESet.mth element.getChildren();
        newAmbient(AmbName : string);
        AmbName.addChild(ChildrenA);
        AmbName.addChild(ChildrenB);
        newSource ("ambName",ambName,ParameterList);
        sources.add(AmbName);
        sources.delete(AESet. nth element);
        sources.delete(AESet. mth element);
        recordReading("AmbName",ParamName,ParameterKind)

Valuations
        [trigger] Attribute=NewValue;
Triggers
        MERGE(AESet)/DECOMPOSE(AESet)
when
        Sources.ParamIter().includes(ParamName) &
        Parameter reading comparisons among N microservice ambients
End_Aspect
```

Figure 2. GranAdapt aspect template specification

A. GranAdapt Aspect and Microservice Ambient Templates

We have defined different templates that provide guidance for architects when defining a microservice ambient and its granularity adaptation behaviour. Architects can then instantiate these templates according to specific microservitization scenarios. A template of the granularity adaptation aspect is outlined in Figure 2, using the Ambient-PRISMA Textual Language that is based on a variant of dynamic logic [19]. The list of architectural elements which a microservice ambient is allowed to monitor is stored in set called *Sources* and the size of the set is stored in *SourceNum*. Each *Source* stores the architectural element located in an ambient, its name and the corresponding *Parameter*s monitored for it. A *Parameter* is characterised by its name, *ParameterKind* and sequence

of recorded values. The *recordReading IMonitor* interface service receives parameters readings from an architectural element inside an ambient implementing this service. The *getReading IMonitor* interface service publishes the latest reading recorded for a parameter of an architectural element. External microservice ambients can request readings for architectural elements that they can not directly communicate with. The italicised terms are variability points in the template, which can be instantiated depending on the followed microservitization scenario.

```
Ambient_Microservice type ReviewFR

Import Granularity Adaptation Aspect GranAdapt
Import Mobility Aspect MobilityAspect
Import Coordination Aspect ACoordination
Import Distribution Aspect ADist
Weavings
        ADist.getLocation(Location) instead MobilityAspect.getParent(Parent);
    GranAdapt.removeChild(AE) after ADistAspect.getLocation(Location);
End_Weavings

Ports
    InMonitorPort: IMonitor;
    ExMonitorPort: IMonitor;
    InCapabilitiesPort: ICapability;
    ECapabilitiesPort: ICapability;
    InServicesPort: ICall;
    EServicesPort: ICall;
    InRoutePort: IGetRoute;
End_Ports
End Ambient_Microservice type ReviewFR
```

Figure 3. Microservice ambient template specification

B. Architectural Configuration Specification

A template of the microservice ambient importing the granularity adaptation aspect is outlined in Figure 3, also using the Ambient-PRISMA Textual Language. This template includes all the invariants a microservice ambient needs to fulfil. The highlighted lines refer to *invGranAdapt* invariant. Because it is a template, Figure 3 does not specify the architectural elements within the microservice ambient or the attachments between them [10]. Moreover, it is assumed that the attachments between a microservice ambient and its children architectural elements is automatically managed by the runtime environment of the microservice ambient [10].

Figure 4 illustrates a possible aspect type that uses the template in Figure 2. Intuitively, it is motivated by clustering simultaneously changing architectural elements within the same microservice ambient. Here the number of attachments (*AttNumber*)across children architectural elements of a microservice ambient (*mem1* and *mem2*) is the indicator of simultaneous change. The parent microservice ambient (*self*) utilises the *getReading* interface service for this monitoring. Where simultaneous change occurs in the number of attachments within *mem1* and *mem2*, this triggers merging *mem1* and *mem2* into a single microservice ambient.

Figure 2 defines merging as a transaction, allowing the rollback of merging and decomposition in case of error. These transactions utilise the interface services from *IMonitor* and establish the changes in the ambient hierarchy required when a child microservice ambient enters or leaves its parent ambient. Crucially, these changes assume that only sibling microservice ambients are merged or decomposed. If a transaction is invoked on non-sibling microservice ambients, the services utilised in the transactions and the triggers in Figure 4 would have been different. The *Valuations* part of Figure 4 updates the list of microservice architectural elements that *self* would have to monitor after merging. This involves removing *mem1* and *mem2* from the *Sources* of *self* and replacing them with the newly formed microservice ambient *cluster*. Once we have the types of our architecture, we

```
Granularity Aspect Clustering using IMonitor

Attributes
Sources: Set (Source);
SourceNum:Integer
....
Transactions
        DECOMPOSE (input AESet: Set(ArchitecturalElement))
        MERGE (input AESet: Set(ArchitecturalElement))
Valuations
        [self.addChild(cluster)] Sources.add(new Source("cluster",cluster,new
Parameter("AttNumber",Integer,[])));
        [self.addChild(cluster)] SourcesNum=SourceNum+1;

        [self.removeChild(mem1)] Sources.delete (mem1);
        [self.removeChild(mem1)] SourcesNum=SourceNum-1;

        [self.removeChild(mem2)] Sources.delete(mem2);
        [self.removeChild(mem2)] SourcesNum=SourceNum-1;
Triggers
        MERGE({mem1,mem2})
when
        Sources.ParamIter().includes("ChangeEvent") &
        getReading(self,"mem1","AttNumber")=getReading(self,"mem2","AttNumber")
End_Aspect
```

Figure 4. Clustering aspect in microservitization scenario

create different configurations by instantiating the elements including ambients and defining the ambient hierarchies. Figure 6.a illustrates a possible instantiation — architectural configuration — of the microservice ambient template. A textual reflection of this graphical architectural configuration is shown in Figure 5. *Movie Review*, *RevInfoReqM* and *RevPolM* are all instances of the microservice ambient *ReviewFR*. The constructor instantiating each microservice ambient instance takes as a parameter its parent's name.

Referring to Figure 6.a, *MovieReview* is the highest level microservice ambient therefore its constructor does not take any parameters. In reality however, this microservice ambient might reside within a larger ambient but for the purpose of illustration we focus on this scope of the architectural configuration. *InfoReqDef1* and *RevPolDef1* are instances of the functional elements residing in *RevInfoReqM* and *RevPolM* respectively. Because the runtime environment maintains the attachments between the microservice ambient and its children architectural elements, only attachments between the children architectural elements need to be explicitly configured. Therefore, only the attachment between *RevIn-*

foReqM and *RevPolM* is explicitly defined. Moreover, this automatic runtime management of attachment ensures that granularity adaptation is invoked reliably —– the attachments are maintained correctly after the adaptation is invoked.

Both the template and configuration specifications are defined at design time according to the microservitization scenario (in this case clustering simultaneous changes together). At runtime, the transactions and valuations of the instantiated aspects and the invariants of the instantiated ambients are executed and managed. The power of the triggers in the granularity adaptation aspect can be extended such that it controls runtime switching across different instances of the granularity adaptation aspect, thereby changing the theme of microservitization at runtime.

```
Architectural_Model_Configuration MovieReviewsConf

New MovieReviews
{
MovieReview = new ReviewFR ();
RevInfoReqM=new ReviewFR (MovieReview);
RevPolM=new ReviewFR (MovieReview);

InfoReqDef1 = new InforReqDef  (``RevInfoReqM'');
RevPolDef1 = new RevPolDef (“RevPolM”);


AttRevInfoRevPol1 = new AttRevInfoRevPol (RevInfoReqM, ESInfoPort,RevPolM, ESPolPort);
```

Figure 5. Possible architectural configuration using microservice ambient instances

VI. EVALUATION

In this section we apply our microservice ambients to model a microservitization scenarios [20], [21]. We then evaluate our modelling approach for flexibility, expressiveness and support for runtime analysis using properties from [11]–[13].

A. Clustering for Localising change

Problem Context: There are cases where sets of functional elements change simultaneously over time. Figure 6.a exemplifies such a scenario. The change events in the *RevInfoReqM* (managing the user input requirements when uploading a review) and *RevPolM* (managing the regulations on the contents of the uploaded reviews) are monitored by the parent *MovieReview* ambient. The attachment between the ExM (ExMonitor) ports of the sub-ambients and the InM (InMonitor) port of the parent ambient enforces this monitoring.

Problem: Localising change and reducing its ripple effects through granularity adaptation.

Main driving forces:

- Reducing logical dependencies across microservice boundaries, thereby enhancing the autonomy of each individual microservice and the maintainability/evolvability of the overall architecture.
- Enhancing the productivity and independence of the team responsible for a microservice.

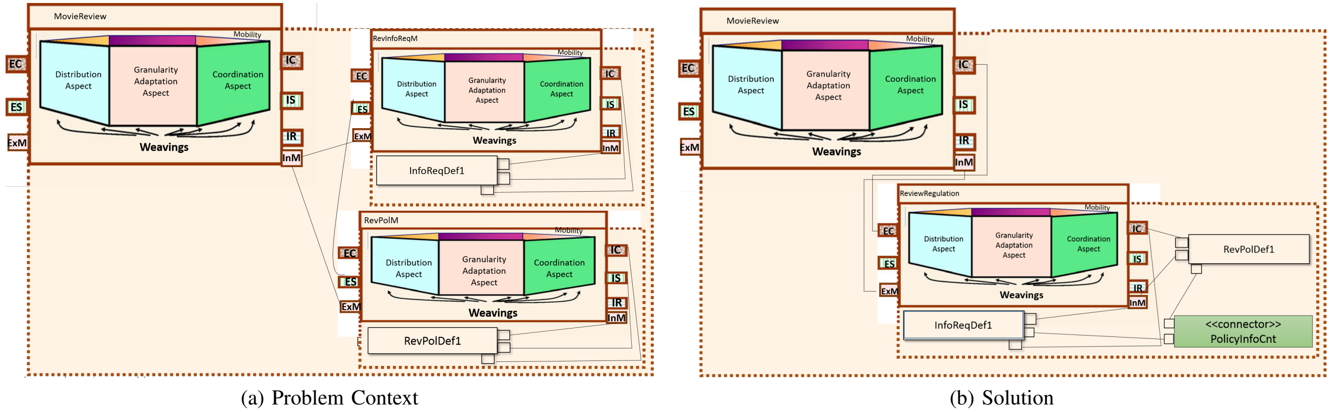
Invoking granularity adaptation: In Figure 6.b, the related functional elements are now encapsulated by a single microservice ambient after the granularity adaptation is applied. The *GranAdapt* aspect definition of this application of the scenario is shown in Figure 4. Reflecting on the number of attachments, there is a total of 9 attachments in Figure 6.a. After invoking the granularity adaptation aspect, there is a total of 8 attachments within the *MovieReview* microservice ambient in Figure 6.b. This reduction in the number of attachments is an indicator of a slight reduction in the logical dependencies when this adaptation is invoked. Although reducing one attachment can seem insignificant at the modelling level, it can translate to a significant reduction in deployment costs at the underlying code level and subsequent reduction in maintenance and testing efforts over time.

Further analysis can help elaborate the merging decision further. Such analysis can optimise for balancing between the reduction in logical dependencies and the accompanied increase in overheads. Examples of overheads include the cost of merging/decomposing the underlying code, additional network link costs (in the case of decomposition), and additional data format translation (in the case of merging).

Logical dependency reduction can enhance the productivity of the team managing the microservice. It particular, the reduction easier for identifying the right expertise that can best deal with the design, development and utility-driven evolution of the microservice, as these concerns to big extent are realised in the “glues” between the microservices — the logical dependencies. Allocating the right expertise potentially reduces long-term social and technical debt in large-scale, distributed systems. This is the case for many microservice applications.

B. Qualitative Evaluation

1) Effectiveness and Expressiveness of Modelling: We use properties from the ADL classification framework in [11] to evaluate the expressiveness and effectiveness of microservice ambients in providing systematic architectural support for modelling microservice boundaries (Requirement 1). ADLs are deemed crucial to architecture-centric modelling. We evaluate the expressiveness and effectiveness of microservice ambients using properties from an ADL classification framework as microservice ambients can essentially support an ADL that realises the concerns of microservices. We evaluate microservice ambients irrespective of the particular application domain and the business competencies in that domain, therefore we use this framework as opposed to the more recent framework presented in [22] which comprises the domain and business aspects of modelling architectures. We elicited the classification framework



(a) Problem Context

(b) Solution

Figure 6. Merging for Change Clustering

properties which are relevant to microservitization to make the evaluation more focussed.

According to [11], the essential elements that an ADL must explicitly model are components, connections, and their architectural configurations. Components (microservice ambients in our case) are the unit of computation in an ADL. Connections in turn model the relationships across these components. Architectural configurations represent the overview of the architecture comprising both components and connections [11]. Architectural configurations therefore depict the impact of design decisions regarding granularity on the overall microservice architecture.

Component modelling:

- **Component Interface**: The interface of a component defines computational commitments a component can make and constraints on its usage [11]. Microservice ambients have commitments regarding the scope of runtime monitoring for each microservitization scenario. A microservice ambient also has commitments on which architectural elements it can control when triggering granularity adaptation. The monitoring commitments are captured by the *Sources* parameter of the granularity adaptation aspect. In Figure 4, the *Sources* parameters only include the children of the microservice ambient importing those aspect instances. For example, the instantiation of the *Sources* parameter in the granularity adaptation aspect of the *MovieReview* microservice ambient in Figure 6.a would include only *RevInfoReqM* and *RevPolM*. On the other hand, the commitments of control are captured by the input to the *MERGE* and *DECOMPOSE* transactions of the granularity adaptation aspect. For example, the input to the *MERGE* transaction in Figure 4 would be *RevInfoReqM* and *RevPolM* referring to Figure 6.a.
- **Component Evolution**: We introduce the granularity adaptation aspect to support boundary evolution of a microservice ambient. Figure 6 demonstrates how the granularity adaptation aspect can enforce evolution of the architecture to optimise for multiple quality drivers. This is a novelty compared to absent or limited support for component evolution in other examined ADLs. MetaH [23] and UniCon [24] for example do not support component evolution thereby they can't be used to reason about dynamic problems such as microservice granularity. Other ADLs such as ACME [25], Rapide [26] and C2 [27] support component evolution by sub-typing, which is a less suitable counterpart to decomposition/merging which we support using transactions in the granularity adaptation aspect.
- **Component non-functional properties**: Localising change addresses the changeability and replaceability non-functional properties of the architecture. Since both properties can be encoded in the granularity adaptation aspect, our contribution supports modelling distinct quality trade-offs related to different levels of microservice granularity. Since microservitization is ultimately a utility-driven exercise [1], modelling non-functional properties is particularly significant to microservitization since it allows reasoning about the impact of different microservitization scenarios on utility.

Connection modelling:

- **Connection types**: Microservice ambients make explicit use of two types of connection: attachments and weavings. Attachments model the relationship across microservice ambients (and other architectural elements). Therefore, attachments are central to analysing the impact of adapting a microservice boundary. Weavings on the other hand enforce reliable granularity adaptation. The weaving relationship we introduce between the granularity adaptation aspect and the distribution aspect ensures that a granularity adaptation triggered by the granularity adaptation aspect is reflected in the hierarchy of ambients managed by the distribution aspect.

Architectural Configuration Modelling:

- **Configuration Understandability**: The graphical support for microservice ambients enables visualising the overall configuration. In particular, the graphical support is detailed

enough to express granularity adaptation in action. However, it is still abstract enough to hide implementation details of this adaptation. On the other hand, the textual support for modelling the configuration (Figure 5) complements the graphical support and sharpens the overall understandability of the modelling approach.

- **Configuration compositionality and scalability**: A compositional ADL supports describing architectural configurations at different levels of detail [11]. The ambient approach in general allows hierarchical composition of architectural elements; it supports compositionality. Support for compositionality facilitates iterative analysis of microservitization scenarios. For example, the internals of the *ReviewRegulation* ambient in Figure 6.b can be further analysed to invoke granularity adaptation within this microservice ambient.
- **Configuration refinement and traceability**: An ADL must enable consistent mapping between its graphical notation and an executable system [11]. An aspect specification can be automatically mapped to a runnable modelling construct using AMBIENT-PRISMANET middleware [10] rendering an implementable runtime model of the solution space of this decision problem.
- **Configuration heterogeneity**: It is essential for ADLs to facilitate modelling architectures that employ varying technological choices (e.g., different programming languages) [11]. The aspect-oriented nature of microservice ambients supports such heterogeneous modelling. Crucially, the granularity adaptation aspect captures granularity adaptation behaviour independent of the technologies used within the architectural elements involved in Sthis behaviour. Support for modelling heterogeneous configurations aligns with the decentralised governance enforced by microservitization [1].
- **Configuration evolvability and dynamism**: ADLs need to support incremental addition, removal, replacement, and reconnection of components and connections in a configuration [11]. The *MERGE* and *DECOMPOSE* transactions of the granularity adaptation aspect provide reliable support for this dynamism — a novelty in our work. ADLs such as Darwin [28] and [26] support reconnection, addition and removal of components and connectors. However, the reliability of these changes are not ensured.
- **Configuration constraints**: An ADL needs to model configuration-wide dependencies in addition to component constraints [11]. Configuration constraints are supported by attachments, since they depict the dependencies across microservice ambients. Support for defining configuration constraints is essential to capturing the impact of different microservitization drivers (e.g., enhancing team productivity and architecture changeability) on the overall architecture.
- **Configuration non-functional properties**: Analogous to component non-functional properties, an ADL needs to support representing configuration-wide non-functional properties [11]. We support this by providing a template of the microservice ambient. Each architectural configuration in turn instantiates this template according to the required non-functional properties.

2) Facilitating Design Time and Runtime Analysis: We use properties from [12], [13] to evaluate the support that microservice ambients provide for runtime analysis of granularity (Requirement 2). In this section we illustrate the significance of each feature in relation to runtime reasoning about the microservice granularity problem and how it is supported in the microservice ambient approach.

- **Location:** Location refers to space where an architectural element is allowed to move. This is significant to defining the scope of the solution space in the granularity problem. Location is enforced by the *Sources* parameter of the granularity adaptation aspect. The architectural elements that a microservice ambient monitors determines the ambient's scope of knowledge about its runtime environment. This knowledge in turn confines the space in which the children of a microservice ambient can be decomposed or merged. This support for location is novel to microservice ambients. Although most ADLs examined (e.g. [25], [27], [28]) support logical location by allowing hierarchical compositions of components and connectors, they do not capture explicitly the space in the hierarchy where these components are allowed to move, which is significant to the microservice granularity problem.
- **Location-awareness:** Awareness of location is supported by the distribution aspect imported by the microservice ambient. In addition, the weaving relation between the granularity adaptation aspect and the distribution aspect ensures that location awareness is maintained after granularity adaptation decisions are executed. Location-awareness is a co-requisite for supporting location; only if a microservice ambient is aware of its position can it reason about the candidate solution space for granularity adaptation.
- **Unit of Mobility:** This is the most central feature to the granularity adaptation problem. Microservice ambients by definition facilitate runtime reasoning about the unit of mobility in a microservice architecture. The unit of mobility refers to the "smallest entity of a model that is allowed to move [12, p.2]." Support for component evolution and configuration dynamism in the previous subsection are both pre-requisites for reasoning about the unit of mobility. In turn, it is the granularity adaptation aspect which defines the unit of mobility when decomposing or merging a microservice ambient. This explicit support for unit of mobility is a novelty in this work compared to ADLs such as ACME [25]. In these ADLs, invoking a modification to an architectural configuration involves re-defining both components and connectors. In microservice ambients, it is the runtime environment which implicitly handles the implications of granularity adaptation, allowing the unit of mobility to be captured clearly as inputs to the decompose and merge transactions (Figure 4).

VII. Related Work

Microservice adopters have modelled microservice architectures using several concepts. Here we summarise the concepts we examined, comparing and contrasting them to the microservice ambient concept. The most common microservices modelling technique is domain-driven modelling [16], [17]. The constructs of a domain model are the different areas of concern within the system, where the refinement of the boundaries between these areas is incremental, triggered by knowledge updates about the system domain. Microservice ambients provide a modelling approach for microservice computation boundaries analogous to domain-driven modelling for the system domain. Moreover, they provide support for runtime reasoning about adapting these boundaries with the granularity adaptation aspect.

In [29],a static microservice design model is proposed which is made up of 5 dimensions: culture, organisation, processes and tools, individual service ("micro-") design and overall architecture ("macro-") design. Moreover, it is possible to create a hierarchy of design models. Microservice ambients enrich this design model with support for runtime analysis. In particular, microservice ambients support modelling granularity of the "micro-design" and facilitate runtime reasoning about the "macro-design".

The hexagonal architecture presented in [30] focuses on the user-facing interfaces. For each business functionality, several adaptors and interfaces are modelled depending on the user type. Consequently, each of these adaptors and interfaces is mapped to a separate microservice. The hexagonal architecture supports adaptability by adding and removing adaptors and/or interfaces. The microservice ambients on the other hand support adaptability in terms of changing the scope of a business functionality by decomposing and merging them. Our support for merging and decomposition gives more insight into the behaviour of each microservice ambient than with the hexagonal modelling approach.

Microservice functionality is modelled in [31] as messages which a microservice sends and/or receives. Different communication patterns for different message syntaxes define the behaviour of a microservice sending or receiving the message. However, modelling the concerns of each microservice using aspects provides a more powerful, in-depth definition of each microservice's behaviour than the light weight message-based approach.

VIII. Conclusion and Future Work

In this paper we provide an architecture-centric approach to model microservice granularity. In particular, we extend the aspect-oriented meta-modelling approach of ambients with ***microservice ambients*** — a modelling concept that treats boundaries as an adaptable first-class entity of microservices. We demonstrate the use and significance of our approach by applying it to a microservitization scenario of a hypothetical online movie subscription-based system. The application shows that microservice ambients can expressively capture microservitization scenarios with distinct QoS trade-offs — driving forces. Additionally, we evaluate the microservice ambients using properties from ADL classification frameworks. The evaluation shows the potential of microservice ambients as an ADL for microservices.The evaluation highlights how microservice ambients support analysis, evolution and mobility/location awareness.

In our short term future work we aim to map the microservice ambients (and their constituents) to concrete microservice source code for the online movie subscription-based system to assess the practicality of microservice ambients as an ADL for microservices.

Due to their power in capturing distinct scenarios, microservice ambients can provide the primitives for devising a catalogue of granularity adaptation patterns, each mapping to different quality trade-offs. A related interesting research direction is mapping each quality to underlying runtime metrics reflecting it. For example, attachment number is used as metric of logical dependency reduction in the change clustering scenario — "pattern". The total number of architectural elements, the ratio of functional elements to microservice ambients and the number of attachments between functional elements are further examples of metrics that can be mapped to the logical dependency reduction quality.Furthermore, the granularity adaptation aspect can be used to identify use cases of conflict among patterns. For each conflict use case, related patterns can resolve the conflict. One direction of future research therefore is to cross-reference patterns into a directed graph that the research community can use to form a pattern language for microservices. There are already promising attempts at this in the industrial community [32].

References

[1] S.Hassan and R.Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," in *13th IEEE International Conference on Services Computing (SCC)*, San Francisco, USA, jun 2016.

[2] K.Probst and J.Becker, "Engineering trade-offs and the netflix api re-architecture," http://techblog.netflix.com/2016/08/engineering-trade-offs-and-netflix-api.html, aug 2016.

[3] T.Wagner, "Microservices without the servers," sep 2015. [Online]. Available: https://aws.amazon.com/blogs/compute/microservices-without-the-servers/

[4] E.Reinhold, "Lessons learned on uber's journey into microservices," jul 2016. [Online]. Available: https://www.infoq.com/presentations/uber-darwin\?utm_campaign=infoq_content\&utm_source=infoq\&utm_medium=feed\&utm_term=Microservices

[5] T.Huston, "What is microservices architecture?" https://smartbear.com/learn/api-design/what-are-microservices/.

[6] Z.Dehghani, "Zhamak dehghani real world microservices: Lessons from the frontline," https://youtu.be/hsoovFbpAoE, Youtube, feb 2015.

[7] N.Alshuqayran, N.Ali, and R.Evans, "A systematic mapping study in microservice architecture," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications*, 2016.

[8] M.Fowler, "Microservicepremium," may 2015. [Online]. Available: http://martinfowler.com/bliki/MicroservicePremium.html

[9] L.Cardelli, *Abstractions for Mobile Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 51–94. [Online]. Available: http://dx.doi.org/10.1007/3-540-48749-2_4

[10] N.Ali, I.Ramos, and C.Sols, "Ambient-prisma: Ambients in mobile aspect-oriented software architecture," *Journal of Systems and Software*, vol. 83, no. 6, pp. 937 – 958, 2010, software Architecture and Mobility. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121209003161

[11] N.Medvidovic and R. N.Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, Jan. 2000. [Online]. Available: http://dx.doi.org/10.1109/32.825767

[12] N.Ali, C.Solís, and I.Ramos, "Comparing architecture description languages for mobile software systems," in *Proceedings of the 1st International Workshop on Software Architectures and Mobility*, ser. SAM '08. New York, NY, USA: ACM, 2008, pp. 33–38. [Online]. Available: http://doi.acm.org/10.1145/1370888.1370897

[13] G.-C.Roman, G. P.Picco, and A. L.Murphy, "Software engineering for mobility: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 241–258. [Online]. Available: http://doi.acm.org/10.1145/336512.336567

[14] N.Ali *et al.*, *Mobile Ambients in Aspect-Oriented Software Architectures*. Boston, MA: Springer US, 2007, pp. 37–48. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39388-9_4

[15] "The promise repository of empirical software engineering data," http://openscience.us/repo. North Carolina State University, Department of Computer Science, 2015.

[16] S.Newman, *Building Microservices*, 1st ed. O'Reilly Media, feb 2015.

[17] E. J.Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

[18] N.Ali and C.Solis, "Self-adaptation to mobile resources in service oriented architecture," in *2015 IEEE International Conference on Mobile Services*. IEEE, 2015, pp. 407–414.

[19] J.Pérez *et al.*, "Integrating aspects in software architectures: Prisma applied to robotic tele-operated systems," *Inf. Softw. Technol.*, vol. 50, no. 9-10, pp. 969–990, Aug. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2007.08.007

[20] M.Fowler and J.Lewis, "Microservices a definition of this new architectural term," http://martinfowler.com/articles/microservices.html, March 2014.

[21] M.Nygard, *Release It!: Design and Deploy Production-ready Software*, ser. Pragmatic Bookshelf Series. Ł, 2007. [Online]. Available: https://books.google.co.uk/books?id=md4uNwAACAAJ

[22] N.Medvidovic, "Moving architectural description from under the technology lamppost," in *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, Aug 2006, pp. 2–3.

[23] A.Agrawala, J.Krause, and S.Vestal, "Domain-specific software architectures for intelligent guidance, navigation and control," in *Computer-Aided Control System Design, 1992. (CACSD), 1992 IEEE Symposium on*, Mar 1992, pp. 110–116.

[24] M.Shaw *et al.*, "Abstractions for software architecture and tools to support them," *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, pp. 314–335, Apr. 1995. [Online]. Available: http://dx.doi.org/10.1109/32.385970

[25] D.Garlan, R.Monroe, and D.Wile, "Acme: An architecture description interchange language," in *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '97. IBM Press, 1997, pp. 7–. [Online]. Available: http://dl.acm.org/citation.cfm?id=782010.782017

[26] D. C.Luckham *et al.*, "Specification and analysis of system architecture using rapide," *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, pp. 336–355, Apr. 1995. [Online]. Available: http://dx.doi.org/10.1109/32.385971

[27] N.Medvidovic *et al.*, "Using object-oriented typing to support architectural design in the c2 style," in *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '96. New York, NY, USA: ACM, 1996, pp. 24–32. [Online]. Available: http://doi.acm.org/10.1145/239098.239106

[28] J.Magee *et al.*, "Specifying distributed software architectures," in *Proceedings of the 5th European Software Engineering Conference*. London, UK, UK: Springer-Verlag, 1995, pp. 137–153. [Online]. Available: http://dl.acm.org/citation.cfm?id=645385.651497

[29] I.Nadareishvili *et al.*, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, 2016.

[30] E.Wolff, *Microservices: Flexible Software Architectures*. CreateSpace Independent Publishing Platform, 2016.

[31] R.Rodger, *The Tao of Microservices*. Manning Publications, 2016.

[32] C.Richardson, "A pattern language for microservices," http://microservices.io/patterns/index.html, 2014.