Formal Methods and Patterns for Microservices

Ph.D. Thesis

by Larisa Safina

University of Southern Denmark

Department of Mathematics and Computer Science

Odense, Denmark

October 2019

Academic Advisors

Full Professor **Fabrizio Montesi**, Ph.D.
Department of Mathematics and Computer Science
University of Southern Denmark
Odense, Denmark

Full Professor **Manuel Mazzara**, Ph.D.
Software Engineering Laboratory
Innopolis University
Innopolis, Russia

Assistant Professor **Luís Cruz-Filipe**, Ph.D.
Department of Mathematics and Computer Science
University of Southern Denmark
Odense, Denmark

Assessment Committee

Full Professor **Thomas Hildebrandt**, Ph.D.
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark

Lecturer **Ornela Dardha**, Ph.D.
School of Computing Science
University of Glasgow
Glasgow, UK

Associate Professor **Jacopo Mauro**, Ph.D.
Department of Mathematics and Computer Science
University of Southern Denmark
Odense, Denmark

Contents

English summary

Microservices is an architectural style that organizes an application as a set of highly granular loosely-coupled services. Microservices often use choreography instead of orchestration as a model of coordination. A choreography is a global specification of how endpoints in a concurrent system should communicate. The algorithm of End Point Projection translates a global choreography model to a set of local specifications. The inverse choreography extraction algorithm recreates a global choreography from local specifications.

This thesis investigates the microservices architectural style, its quality model and formal approaches to data communication and querying. It provides three main contributions.

The first contribution is the comparison of microservices against previous engineering approaches and the analysis of their strong and weak sides, which can be exploited for developing solid architecture patterns.

The second is the implementation of the choreography extraction algorithm proposed by Cruz-Filipe et al. and conducting the empirical analysis of its performance. Also, some improvements to the original algorithm were investigated.

The third is the development of TQuery, a framework for document-oriented queries, inspired by MQuery, the sound variant of the MongoDB query language. TQuery was implemented as a sub-language inside of the microservices-based programming language Jolie.

Danish summary

Microservices er en arkitektur, den organiserer en applikation som et sæt af meget granulært, løst koblede services. Microservices bruger koreografi i stedet for orkestrering som en koordinationsmodel. En koreografi er en global specifikation af, hvordan endpunkter i et concurrent system skal kommunikere. Algoritmen til End Point Projection oversætter en global koreografimodel til et sæt lokale specifikationer. Den inverse koreografiske ekstraktionsalgoritme genskaber en global koreografi ud fra lokale specifikationer.

Dette projekt undersøger microservices arkitektur, dens kvalitetsmodel og formelle tilgange til datakommunikation og forespørgsler. Det giver tre hovedbidrag.

Det første bidrag er sammenligningen af microservices med deres arkitektoniske forgængere og analysen af deres stærke og svage sider, der kan udnyttes til at udvikle solide arkitekturmønstre.

Den anden er implementeringen af den koreografiske ekstraktionsalgoritme foreslået af Cruz-Filipe et al. og udføre den empiriske analyse af dens ydeevne. Nogle forbedringer af den originale algoritme blev også undersøgt.

Den tredje er udviklingen af TQuery, en ramme for dokumentorienterede forespørgsler, inspireret af MQuery, den formelt korrekte variant af MongoDB-forespørgselssprog. TQuery blev implementeret som et undersprog inde i det microservices baserede programmeringssprog Jolie.

Relation to published material

Chapter II

Includes a transcript of the paper:

— Dragoni N., Giallorenzo S., Lluch Lafuente A., Mazzara M., Montesi F., Mustafin R., Safina L. (2017) Microservices: yesterday, today, and tomorrow. *Manuel Mazzara; Bertrand Meyer. Present and Ulterior Software Engineering, Springer*, 978-3-319-67425-4.

Chapter III

A paper including results from chapter 3 has been submitted for publication.

Chapter IV

Includes the papers:

— Giallorenzo S., Montesi F., Safina L., Zingaro S. (2019) Ephemeral Data Handling in Microservices. *IEEE International Conference on Services Computing, pp. 234-236*

— Giallorenzo S., Montesi F., Safina L., Zingaro S. (2019) Ephemeral Data Handling in Microservices - technical report.

Acknowledgements

First, I would like to thank my fantastic supervisors Fabrizio Montesi, Manuel Mazzara, and Luís Cruz-Filippe, who spent a lot of time and attention on me (and have never made me cry).

I would like to thank my reviewers Jacopo Mauro, Ornela Dardha, and Thomas Hildebrandt, who have found the time for reading my thesis.

I would like to thank all my friends and colleagues at Innopolis, especially the members of the Software Engineering Laboratory: Alexander Naumchev, Mansur Khazeev, Daniel de Carvalho, Manuel Mazzara, and Bertrand Meyer.

I would like to thank my all my friends and colleagues I've met at SDU and Odense, especially Marco Peressotti, Saverio Giallorenzo, Daniela Monti, Marco Chiarandini, Arthur Zimek, Jacopo Mauro, Luís Cruz-Filippe, Keld Homburg, and Julia Meffe. Thank you for the board games, and "other fun stuff".

My family plays a huge role in my life and has made me who I am (for better or for worse). I would not be here without my parents Galina Safina and Rais Safin, without my sister and friend Asya Safina, and without my grandparents Josefina and Innokentiy Bezdenejnych, Maya Safina and Niyaz Safin. Thank you for your warm and support.

I would like to thank Yunus Zaytaev, with whom we spent a lot of time as classmates, colleagues, and as good friends.

I would like to thank Benedikt Ahrens for organizing the series of UniMath schools, which I had a chance and pleasure to visit during my Ph.D. program.

I would like to thank Alex Tchitchigin for luring me into the magical world of functional programming and type systems. Alex, how could you!

And finally, I would like to thank Svyatoslav Zhirenko, Olga Litvinova, Marina Rodionova, Vasiliy Artemyev, Aliona Kozlova, Ekaterina Loseva, Igor Andrianov, and Oleg Abakumov for appearing in my life.

To Max Talanov

Whenever I told Max that I want to do something that takes courage, he always supported me.

Max is the one to blame for my presence in academia.

Chapter 1

Introduction

The topic of this thesis is the microservices architectural style, focusing on three particular aspects. The first part of it is about the microservices in general, their evolution from SOA, the current state, and the future. Another part is about choreographies, a global model of endpoint coordination in distributed systems, and the algorithm of choreographies extraction. And the last part is about ephemeral data handling approached in the context of microservice-oriented programming language Jolie.

To each of these subjects, we dedicate a chapter in the thesis and a subsection of this introduction to recite briefly the research question we were interested in, and the contributions we have made.

1.1 Microservices and what are they good for

Chapter 2 discusses the past, presence, and future of microservices.

The past of microservices and their predecessor service-oriented architecture [131] is monolithic architecture which combines all components in a single application. This approach had many problems to be addressed, such as the reduced scalability, entanglement of dependencies (leading to inconsistency and difficulties in maintenance), and downtimes caused by changes in functionality that require restarting the application.[1] The first approaches to these problems were the ideas of separation of concerns [80] and Component-based software engineering (CBSE) [176], which became the fertile ground for the emergence of service-oriented architecture (SOA). SOA has introduced the decoupling of system components into services, the orchestration as

[1] Further discussion on the monolith's pros and cons can be found in *"How the monolith betrays the promise of components"* in [167] or in *"Escaping monolithic hell"* in [165]

the components coordination model, and the heterogeneity of implementation. Microservice architectures were the next step, pushing the ideas of CBSE and SOA further. Microservice architectures focus around business capabilities, uses choreographies over orchestration (which helps with problems of bottlenecks or single point of failure [110]), and make each service implement only a limited amount of functionality.

Microservices architecture gives us pervasive distribution, which comes at a price. Programming and managing loosely coupled components are error-prone due to the increased complexity of communications between them. Microservices strongly depend on some core non-functional properties to guarantee software quality, like elasticity, security, and reliability [124]. While some preliminary patterns to deal with some of these aspects have already emerged, e.g., API Gateway and Circuit Breaker, we are still far from understanding the full picture and defining a comprehensive portfolio of patterns that can be used by designers to address quality concerns.

The main contribution of this chapter is an analysis of the microservices architecture formation and the current state of the art, which allows us to make predictions on the future evolution of this pattern. We have also investigated the microservices quality model to identify the quality attributes it supports and violates.

1.2 Choreographies and how to extract them

Choreography is a model of coordination in distributed systems, which is opposed to the orchestration model as it does not require a conductor, "a single component for managing others". Choreography has a global view on a system and the collaboration of its components [79]. Even though the orchestration remains the most popular choice of coordination strategy in distributed systems, choreography is gaining popularity together with the microservices architectural style [150, 191].

The most prominent choreography model is WS-CDL [186]. However, some research observes that its formal justification is not as comprehensive [32] as the other approaches in general [77]. The sound formalization of choreographies remains the actual research topic and includes many

stimulating approaches, including [163, 128, 143], etc. One of the principal algorithms developed was the End Point Projection (EPP). This algorithm projects a choreography specification to the set of specifications for the components of the choreography. In [143] Montesi has proposed the model of *choreographic programming*, which among other contributions, was the first one applying EPP to microservices.

Nonetheless, EPP has its disadvantages intrinsic to its top-down nature. For example, if one of the local specifications has been changed, and the global choreography model needs to be adjusted correspondingly. Or if one of the local specifications reflects the legacy part of the system and can not be projected from the global specification of the choreography, in this case the global specification should be extracted from its (and other components) specifications. In both cases, EPP would be not able to reflect the changes and guarantee the conformity of the global model and local specifications anymore. Therefore we need a way of creating a global choreography specification out of local specifications. There exist several approaches to this problem [126, 57, 43], the last one was advanced in [73]. In this work the authors have defined the extraction process as an algorithm on graphs: building a choreography corresponds to finding a path in a graph whose edges are possible communications between local components. The algorithm works for both synchronous and asynchronous communication semantics.

Chapter 3 continues this line of research. Our main contribution is the implementation of the refined version of the extraction algorithm[2]. We have also tested the implementation against different extraction strategies, which define the order in which actions from local specifications are extracted depending on their type. It helps us to confirm the algorithm efficiency claimed in [73] and to show that it does not depend on the order of actions being extracted.

1.3 Ephemerality and data handling

At the early days of microservices, the absene of state considered to be one of its key properties [1, 149]. Being stateless means to have only ephemeral data storages, which makes a stateless microservice immutable, so we can query it ignoring any context or state. Currently, both

[2]Details on the refinement made can be found in section 3.4

stateless and stateful microservices occur in practice [119], but being stateless can pay off in several cases:

1. Lack of possibility for storing big bulks of data, which is typical for Edge Computing or Internet of Things.
2. Restriction on storing of persistent data, due to legal regulations (e.g GDPR, California Privacy, and Protection law [6], Russian Personal Data Law [19], etc).
3. Other ethical, safety or security reasons, prohibiting storing sensitive data (e.g. patients or customers data in eHealth, retail or banking).

Addressing these concerns leads to the emergence of the new property in data handling, called ephemerality. The idea of this concept is to process the data in real-time but to store as little data as possible [173].

Chapter 4 contributes to this line of research. We have analyzed possible strategies to ephemeral data handling, comparing existent data handling frameworks. Next, we have proposed the formal model of a query language for ephemeral data handling, called TQuery, which is based on the work of Botoeva et al. and their model MQuery [41]. We have also implemented TQuery as a library written in programming language Jolie [145] which has native support for microservices.

Chapter 2

Microservices:

yesterday, today, and tomorrow

Abstract

The microservice architecture is a style inspired by service-oriented computing that has recently started gaining popularity. In this chapter, we review the history of microservices and describe their main features. We also highlight the impact of these features on the quality model with the focus on scalability.

2.1 Introduction

The mainstream languages for development of server-side applications, like Java, C/C++, and Python, provide abstractions to break down the complexity of programs into modules. However, these languages are designed for the creation of single executable artefacts, also called monoliths, and their modularisation abstractions rely on the sharing of resources of the same machine (memory, databases, files). Since the modules of a monolith depend on said shared resources, they are not independently executable.

Definition 2.1.1 (Monolith)**.** A monolith is a software application whose modules cannot be executed independently.

This makes monoliths difficult to use in distributed systems without specific frameworks or ad hoc solutions such as, for example, Network Objects [38], RMI [104] or CORBA [155]. However, even these approaches still suffer from the general issues that affect monoliths; below we list the most relevant ones (we label issues **I**|n):

I|1 large-size monoliths are difficult to maintain and evolve due to their complexity. Tracking down bugs requires long perusals through their code base;

I|2 monoliths also suffer from the "dependency hell" [120], in which adding or updating libraries results in inconsistent systems that do not compile/run or, worse, misbehave;

I|3 any change in one module of a monolith requires rebooting the whole application. For large-sized projects, restarting usually entails considerable downtimes, hindering development, testing, and the maintenance of the project;

I|4 deployment of monolithic applications is usually sub-optimal due to conflicting requirements on the constituent models' resources: some can be memory-intensive, others computational-intensive, and others require ad-hoc components (e.g., SQL-based rather than graph-based databases). When choosing a deployment environment, the developer must compromise with a one-size-fits-all configuration, which is either expensive or sub-optimal with respect to the individual modules;

I|5 monoliths limit scalability. The usual strategy for handling increments of inbound requests is to create new instances of the same application and to split the load among said instances.

However, it could be the case that the increased traffic stresses only a subset of the modules, making the allocation of the new resources for the other components inconvenient;

I|6 monoliths also represent a technology lock-in for developers, which are bound to use the same language and frameworks of the original application.

The microservices architectural style [90] has been proposed to cope with such problems. In our definition of microservice, we use the term "cohesive" [78, 113, 37, 47, 25] to indicate that a service implements only functionalities strongly related to the concern that it is meant to model.

Definition 2.1.2 (Microservice)**.** A microservice is a cohesive, independent process interacting via messages.

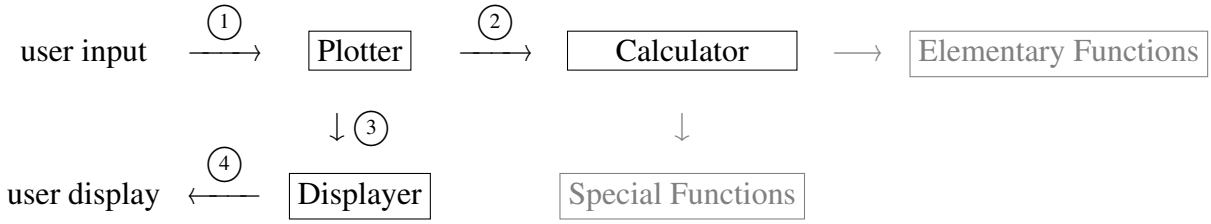
As an example, consider a service intended to compute calculations. To call it a microservice, it should provide arithmetic operations requestable via messages, but it should not provide other (possibly loosely related) functionalities like plotting and displaying of functions.

From a technical point of view, microservices should be independent components conceptually deployed in isolation and equipped with dedicated memory persistence tools (e.g., databases). Since all the components of a microservice architecture are microservices, its distinguishing behaviour derives from the composition and coordination of its components via messages.

Definition 2.1.3 (Microservice Architecture)**.** A microservice architecture is a distributed application where all its modules are microservices.

To give an example of a microservice architecture, let us assume that we want to provide a functionality that plots the graph of a function. We also assume the presence of two microservices: Calculator and Displayer. The first is the calculator microservice mentioned above, the second renders and displays images. To fulfil our goal, we can introduce a new microservice, called Plotter, that orchestrates Calculator to calculate the shape of the graph and that invokes Displayer to render the calculated shape. Below, we report (in black) a depiction of the workflow of such a

microservice architecture.



The developers of the architecture above can focus separately on implementing the basic microservice functionalities, i.e., the Calculator and the Displayer. Finally, they can implement the behaviour of the distributed application with the Plotter that (1) takes the function given by a user, (2) interacts with the Calculator to compute a symbolic representation of the graph of the function, and finally (3) requests the Displayer to show the result back to the user (4). To illustrate how the microservice approach scales by building on pre-existing microservice architectures, in the figure above we drew the Calculator orchestrating two extra microservices (in grey) that implement mathematical Elementary and Special Functions.

The microservice architectural style does not favour or forbid any particular programming paradigm. It provides a guideline to partition the components of a distributed application into independent entities, each addressing one of its concerns. This means that a microservice, provided it offers its functionalities via message passing, can be internally implemented with any of the mainstream languages cited in the beginning of this section.

The principle of microservice architectures assists project managers and developers: it provides a guideline for the design and implementation of distributed applications. Following this principle, developers focus on the implementation and testing of a few, cohesive functionalities. This holds also for higher-level microservices, which are concerned with coordinating the functionalities of other microservices.

We conclude this section with an overview, detailed in greater depth in the remainder of the paper, on how microservices cope with the mentioned issues of monolithic applications (below, **S**|n is a solution to issue **I**|n).

S|1 microservices implement a limited amount of functionalities, which makes their code base small and inherently limits the scope of a bug. Moreover, since microservices are

independent, a developer can directly test and investigate their functionalities in isolation with respect to the rest of the system;

S|2 it is possible to plan gradual transitions to new versions of a microservice. The new version can be deployed "next" to the old one and the services that depend on the latter can be gradually modified to interact with the former. This fosters continuous integration [89] and greatly eases software maintenance;

S|3 as a consequence of the previous item, changing a module of a microservice architecture does not require a complete reboot of the whole system. The reboot regards only the microservices of that module. Since microservices are small in size, programmers can develop, test, and maintain services experiencing only very short re-deployment downtimes;

S|4 microservices naturally lend themselves to containerisation [137], and developers enjoy a high degree of freedom in the configuration of the deployment environment that best suits their needs (both in terms of costs and quality of service);

S|5 scaling a microservice architecture does not imply a duplication of all its components and developers can conveniently deploy/dispose instances of services with respect to their load [92];

S|6 the only constraint imposed on a network of interoperating microservices is the technology used to make them communicate (media, protocols, data encodings). Apart from that, microservices impose no additional lock-in and developers can freely choose the optimal resources (languages, frameworks, etc.) for the implementation of each microservice.

In the remainder of this paper, in § 2.2, we give a brief account of the evolution of distributed architectures until their recent incarnation in the microservice paradigm. Then, we detail the problems that microservices can solve and their proposed solutions in the form of microservice architectures. In § 2.3, we detail the current solutions for developing microservice architectures and how microservices affect the process of software design, development, testing, and maintenance. In § 2.4 we discuss the open challenges and the desirable tools for programming microservice architecture. In § 2.5 we draw overall conclusions.

2.2 Yesterday

Architecture is what allows systems to evolve and provide a certain level of service throughout their life-cycle. In software engineering, architecture is concerned with providing a bridge between system functionality and requirements for quality attributes that the system has to meet. Over the past several decades, software architecture has been thoroughly studied, and as a result software engineers have come up with different ways to compose systems that provide broad functionality and satisfy a wide range of requirements. In this section, we provide an overview of the work on software architectures from the early days to the advent of microservices.

2.2.1 From the early days to Object-oriented design patterns

The problems associated with large-scale software development were first experienced around the 1960s [48]. The 1970s saw a huge rise of interest from the research community for software design and its implications on the development process. At the time, the design was often considered as an activity not associated with the implementation itself and therefore requiring a special set of notations and tools. Around the 1980s, the full integration of design into the development processes contributed towards a partial merge of these two activities, thus making it harder to make neat distinctions.

References to the concept of software architecture also started to appear around the 1980s. However, a solid foundation on the topic was only established in 1992 by Perry and Wolf [160]. Their definition of software architecture was distinct from software design, and since then it has generated a large community of researchers studying the notion and the practical applications of software architecture, allowing the concepts to be widely adopted by both industry and academia.

This spike of interest contributed to an increase in the number of existing software architecture patterns (or generally called *styles*), so that some form of classification was then required. This problem was tackled in one of the most notable works in the field, the book "Software Architecture: Perspectives on an Emerging Discipline" by Garlan and Shaw [172]. Bosch's work [40] provides a good overview of the current research state in software engineering and

architecture. Since its appearance in the 1980s, software architecture has developed into a mature discipline making use of notations, tools, and several techniques. From the pure, and occasionally speculative, realm of academic basic research, it has made the transition into an element that is essential to industrial software construction.

The advent and diffusion of object-orientation, starting from the 1980s and in particular in the 1990s, brought its own contribution to the field of Software Architecture. The classic by Gamma et al. [98] covers the design of object-oriented software and how to translate it into code presenting a collection of recurring solutions, called patterns. This idea is neither new nor exclusive to Software Engineering, but the book is the first compendium to popularize the idea on a large scale. In the pre-Gamma era patterns for OO solutions were already used: a typical example of an architectural design pattern in object-oriented programming is the Model-View-Controller (MVC) [88], which has been one of the seminal insights in the early development of graphical user interfaces.

2.2.2 Service-oriented Computing

Attention to separation of concerns has recently led to the emergence of the so-called Component-based software engineering (CBSE) [177], which has given better control over design, implementation and evolution of software systems. The last decade has seen a further shift towards the concept of service first [184] and the natural evolution to microservices afterwards.

Service-Oriented Computing (SOC) is an emerging paradigm for distributed computing and e-business processing that finds its origin in object-oriented and component computing. It has been introduced to harness the complexity of distributed systems and to integrate different software applications [131]. In SOC, a program — called a service — offers functionalities to other components, accessible via message passing. Services decouple their interfaces (i.e. how other services access their functionalities) from their implementation. On top of that, specific workflow languages are then defined in order to orchestrate the complex actions of services (e.g. WS-BPEL [154]). These languages share ideas with some well-known formalisms from concurrency theory, such as CCS and the π-calculus [138, 139]. This aspect fostered the development of formal models for better understanding and verifying service interactions,

ranging from foundational process models of SOC [133, 107, 130] to theories for the correct composition of services [45, 114, 115]. In [192] a classification of approaches for business modeling puts this research in perspective.

The benefits of service-orientation are:

- **Dynamism** - New instances of the same service can be launched to split the load on the system;
- **Modularity and reuse** - Complex services are composed of simpler ones. The same services can be used by different systems;
- **Distributed development** - By agreeing on the interfaces of the distributed system, distinct development teams can develop partitions of it in parallel;
- **Integration of heterogeneous and legacy systems** - Services merely have to implement standard protocols to communicate.

2.2.3 Second generation of services

The idea of componentization used in service-orientation can be partially traced back to the object-oriented programming (OOP) literature; however, there are peculiar differences that led to virtually separate research paths and communities. As a matter of fact, SOC at the origin was - and still is - built on top of OOP languages, largely due to their broad diffusion in the early 2000s. However, the evolution of objects into services, and the relative comparisons, has to be treated carefully since the first focus on encapsulation and information is hidden in a *shared-memory* scenario, while the second is built on the idea of independent deployment and *message-passing*. It is therefore a paradigm shift, where both the paradigms share the common idea of componentization. The next step is adding the notion of *business capability* and therefore focusing analysis and design on it so that the overall system architecture is determined on this basis.

The first "generation" of service-oriented architectures (SOA) defined daunting and nebulous requirements for services (e.g., discoverability and service contracts), and this hindered the

adoption of the SOA model. Microservices are the second iteration on the concept of SOA and SOC. The aim is to strip away unnecessary levels of complexity in order to focus on the programming of simple services that effectively implement a single functionality. Like OO, the microservices paradigm needs ad-hoc tools to support developers and naturally leads to the emergence of specific design patterns [169]. First and foremost, languages that embrace the service-oriented paradigm are needed (instead, for the most part, microservice architectures still use OO languages like Java and Javascript or functional ones). The same holds for the other tools for development support like testing suites, (API) design tools, etc.

2.3 Today

The microservices architecture appeared lately as a new paradigm for programming applications by means of the composition of small services, each running its own processes and communicating via light-weight mechanisms. This approach has been built on the concepts of SOA [131] brought from crossing-boundaries workflows to the application level and into the applications architectures, i.e. its Service-Oriented Architecture and Programming from the large to the small.

The term "microservices" was first introduced in 2011 at an architectural workshop as a way to describe the participants' common ideas in software architecture patterns [90]. Until then, this approach had also been known under different names. For example, Netflix used a very similar architecture under the name of Fine grained SOA [188].

Microservices now are a new trend in software architecture, which emphasises the design and development of highly maintainable and scalable software. Microservices manage growing complexity by functionally decomposing large systems into a set of independent services. By making services completely independent in development and deployment, microservices emphasise loose coupling and high cohesion by taking modularity to the next level. This approach delivers all sorts of benefits in terms of maintainability, scalability and so on. It also comes with a bundle of problems that are inherited from distributed systems and from SOA, its predecessor. The Microservices architecture still shows distinctive characteristics that blend into something

unique and different from SOA itself:

- **Size** - The size is comparatively small wrt. a typical service, supporting the belief that the architectural design of a system is highly dependent on the structural design of the organization producing it. Idiomatic use of the microservices architecture suggests that if a service is too large, it should be split into two or more services, thus preserving granularity and maintaining focus on providing only a single business capability. This brings benefits in terms of service maintainability and extendability.
- **Bounded context** - Related functionalities are combined into a single business capability, which is then implemented as a service.
- **Independency** - Each service in microservice architecture is operationally independent from other services and the only form of communication between services is through their published interfaces.

The key system characteristics for microservices are:

- **Flexibility** - A system is able to keep up with the ever-changing business environment and is able to support all modifications that is necessary for an organisation to stay competitive on the market
- **Modularity** - A system is composed of isolated components where each component contributes to the overall system behaviour rather than having a single component that offers full functionality
- **Evolution** - A system should stay maintainable while constantly evolving and adding new features

The microservices architecture gained popularity relatively recently and can be considered to be in its infancy since there is still a lack of consensus on what microservices actually are. M. Fowler and J. Lewis provide a starting ground by defining principal characteristics

of microservices [90]. S. Newman [150] builds upon M. Fowler's article and presents recipes and best practices regarding some aspects of the aforementioned architecture. L. Krause in his work [124] discusses patterns and applications of microservices. A number of papers has also been published that describe details of design and implementation of systems using microservices architecture. For example, the authors of [127] present development details of a new software system for Nevada Research Data Center (NRDC) using the microservices architecture. M. Rahman and J. Gao in [99] describe an application of behaviour-driven development (BDD) to the microservices architecture in order to decrease the maintenance burden on developers and encourage the usage of acceptance testing.

2.3.1 Teams

Back in 1968, Melvin Conway proposed that an organisation's structure, or more specifically, its communication structure constrains a system's design such that the resulting design is a copy of the organisation's communication patterns [69]. The microservices approach is to organise cross-functional teams around services, which in turn are organised around business capabilities [90]. This approach is also known as "you build, you run it" principle, first introduced by Amazon CTO Werner Vogels [102]. According to this approach, teams are responsible for full support and development of a service throughout its lifecycle.

2.3.2 Total automation

Each microservice may represent a single business capability that is delivered and updated independently and on its own schedule. Discovering a bug and or adding a minor improvement do not have any impact on other services and on their release schedule (of course, as long as backwards compatibility is preserved and a service interface remains unchanged). However, to truly harness the power of independent deployment, one must utilise very efficient integration and delivery mechanisms. This being said, microservices are the first architecture developed in the post-continuous delivery era and essentially microservices are meant to be used with continuous delivery and continuous integration, making each stage of delivery pipeline automatic. By using automated continuous delivery pipelines and modern container tools, it is possible to

deploy an updated version of a service to production in a matter of seconds [132], which proves to be very beneficial in rapidly changing business environments.

2.3.3 Choreography over orchestration

As discussed earlier, microservices may cooperate in order to provide more complex and elaborate functionalities. There are two approaches to establish this cooperation – orchestration [134] and choreography [159]. Orchestration requires a conductor – a central service that will send requests to other services and oversee the process by receiving responses. Choreography, on the other hand, assumes no centralisation and uses events and publish/subscribe mechanisms in order to establish collaboration. These two concepts are not new to microservices, but rather are inherited from the SOA world where languages such as WS-BPEL [154] and WS-CDL [185] have long represented the major references for orchestration and choreography respectively (with vivid discussions between the two communities of supporters). Apart from them, there are other industry standards used in the context of microservices and choreographies. BPMN [42] is a widely-used business process modeling standard, successfully adopted to be used for microservices and choreographies. Collaboration diagrams in UML can be used to express choreography models [50].

Prior to the advent of microservices and at the beginning of the SOA's hype in particular, orchestration was generally more popular and widely adopted, due to its simplicity of use and easier ways to manage complexity. However, it clearly leads to service coupling and uneven distribution of responsibilities, and therefore some services have a more centralising role than others. Microservices' culture of decentralisation and the high degrees of independence represents instead the natural application scenario for the use of choreography as a means of achieving collaboration. This approach has indeed recently seen a renewed interest in connection with the broader diffusion of microservices in what can be called the second wave of services.

2.3.4 Impact on quality and management

In order to understand the microservices architecture better, to know its strength and limitations, we need to address its quality model. Here we present the short analysis on some of the most important quality attributes.

Scalability Scalability is the key quality attribute in microservices that is supported by design. We found it one of the most peculiar attributes concerning microservices since it participates in multiple quality-related trade-offs and has not trivial effect on some other quality attributes: e.g. improving or suppressing scalability of microservices-based system can impact on performance, availability both positively and negatively. On the one hand, natural distribution enforced with scalability can make the system perform better due to e.g. the efficient parallelization of tasks between microservices. Making the system more scalable will also improve availability since several copies of the same service might be available. On the other hand, having multiple microservices will inevitably deteriorate the performance in the sense of time behavior by increasing network latency. Also running multiple copies of a microservice hosted on the same machine might undermine their availability since the host can simply fail to effectively provide enough resources for each copy.

In general improving system's scalability worsens such qualities as deployment, administration, monitoring, and security by adding extra overhead. For more information on the relation between microservices and scalability, the reader may refer to [83].

Availability Availability is a major concern in microservices as it directly affects the success of a system. Given services independence, the whole system availability can be estimated in terms of the availability of the individual services that compose the system. Even if a single service is not available to satisfy a request, the whole system may be compromised and experience direct consequences. If we take service implementation, the more fault-prone a component is, the more frequently the system will experience failures. One would argue that small-size components lead to a lower fault density. However, it has been found by Hatton [109] and by Compton and Withrow [68] that small-size software components often have a very high fault density. On the other hand, El Emam *et al.* in their work [84] found that as size increases, so does a component's

fault proneness. Microservices are prevented from becoming too large as idiomatic use of the microservices architecture suggests that, as a system grows larger, microservices should be prevented from becoming overly complex by refining them into two or more different services. Thus, it is possible to keep optimal size for services, which may theoretically increase availability. On the other hand, spawning an increasing number of services will make the system fault-prone on the integration level, which will result in decreased availability due to the large complexity associated with making dozens of services instantly available.

Reliability Given the distributed nature of the microservices architecture, particular attention should be paid to the reliability of message-passing mechanisms between services and to the reliability of the services themselves. Building the system out of small and simple components is also one of the rules introduced in [164], which states that in order to achieve higher reliability one must find a way to manage the complexities of a large system: building things out of simple components with clean interfaces is one way to achieve this. The greatest threat to microservices reliability lies in the domain of integration and therefore when talking about microservices reliability, one should also mention integration mechanisms. One example of this assumption being false is using a network as an integration mechanism and assuming network reliability is one of the first fallacies of distributed computing [168]. Therefore, in this aspect, microservices reliability is inferior to the applications that use in-memory calls. It should be noted that this downside is not unique only to microservices and can be found in any distributed system. When talking about messaging reliability, it is also useful to remember that microservices put restrictions on integration mechanisms. More specifically, microservices use integration mechanisms in a very straightforward way - by removing all functionality that is not related to the message delivering and focusing solely on reliable message delivery.

Maintainability By nature, the microservices architecture is loosely coupled, meaning that there is a small number of links between services and services themselves being independent. This greatly contributes to the maintainability of a system by minimising the costs of modifying services, fixing errors or adding new functionality. Despite all efforts to make a system as maintainable as possible, it is always possible to spoil maintainability by writing obscure and counterintuitive code [34]. As such, another aspect of microservices that can lead to increased

maintainability is the above mentioned "you build it, you run it" principle, which leads to better understanding a given service, its business capabilities and roles [86, 67].

Performance The prominent factor that negatively impacts performance in the microservices architecture is communication over a network. The network latency is much greater than that of memory. This means that in-memory calls are much faster to complete than sending messages over the network. Therefore, in terms of communication, the performance will degrade compared to applications that use in-memory call mechanisms. Restrictions that microservices put on size also indirectly contribute to this factor. In more general architectures without size-related restrictions, the ratio of in-memory calls to the total number of calls is higher than in the microservices architecture, which results in less communication over the network. Thus, the exact amount of performance degradation will also depend on the system's interconnectedness. As such, systems with well-bounded contexts will experience less degradation due to looser coupling and fewer messages sent.

Security In any distributed system security becomes a major concern. In this sense, microservices suffer from the same security vulnerabilities as SOA [35]. As microservices use REST mechanism and XML with JSON as main data-interchange formats, particular attention should be paid to providing security of the data being transferred. This means adding additional overhead to the system in terms of additional encryption functionality. Microservices promote service reuse, and as such it is natural to assume that some systems will include third-party services. Therefore, an additional challenge is to provide authentication mechanisms with third-party services and ensure that the sent data is stored securely. In summary, microservices' security is impacted in a rather negative manner because one has to consider and implement additional security mechanisms to provide additional security functionality mentioned above.

Testability Since all components in a microservices architecture are independent, each component can be tested in isolation, which significantly improves component testability compared to monolithic architecture. It also allows to adjust the scope of testing based on the size of changes. This means that with microservices it is possible to isolate parts of the system that changed and parts that were affected by the change and to test them independently from the rest of the system.

Integration testing, on the other hand, can become very tricky, especially when the system that is being tested is very large, and there are too many connections between components. It is possible to test each service individually, but anomalies can emerge from collaboration of a number of services.

2.4 Tomorrow

Microservices are so recent that we can consider their exploration to have just begun. In this section, we discuss interesting future directions that we envision will play key roles in the advancement of the paradigm.

The greatest strength of microservices comes from pervasive distribution: even the internal components of software are autonomous services, leading to loosely coupled systems and the other benefits previously discussed. However, from this same aspect (distribution) also comes its greatest weakness: programming distributed systems is inherently harder than monoliths. We now have to think about new issues. Some examples are: how can we manage changes to a service that may have side-effects on the other services that it communicates with? How can we prevent attacks that exploit network communications?

2.4.1 Dependability

There are many pitfalls that we need to keep in mind when programming with microservices. In particular, preventing programming errors is hard. Consequently, building dependable systems is challenging.

Interfaces Since microservices are autonomous, we are free to use the most appropriate technology for the development of each microservice. A disadvantage introduced by this practice is that different technologies typically have different means of specifying contracts for the composition of services (e.g., interfaces in Java, or WSDL documents in Web Services [65]). Some technologies do not even come with a specification language and/or a compatibility checker of microservices (Node.js, based on JavaScript, is a prime example).

Thus, where do we stand? Unfortunately, the current answer is informal documentation. Most services come with informal documents expressed in natural language that describe how clients should use the service. This makes the activity of writing a client very error-prone, due to potential ambiguities. Moreover, we have no development support tools to check whether service implementations actually implement their interfaces correctly.

As an attempt to fix this problem, there are tools for the formal specification of message types for data exchange, which one can use to define service interfaces independently of specific technologies. Then, these technology-agnostic specifications can be either compiled to language-specific interfaces — e.g., compiling an interface to a Java type — or used to check for well-typedness of messages (wrt. interfaces and independently of the transport protocol). Examples of tools offering these methodologies are Jolie [146, 31, 108], Apache Thrift [162], and Google's Protocol Buffers [182]. However, it is still unclear how to adapt tools to implement the mechanical checking (at compile or execution time) of messages for some widespread architectural styles for microservices, such as REST [87], where interfaces are constrained to a fixed set of operations and actions are expressed on dynamic resource paths. A first attempt at bridging the world of technology-agnostic interfaces based on operations and REST is presented in [144], but checking for the correctness of the binding information between the two is still left as a manual task to the programmer. Another, and similar, problem is trying to apply static type checking to dynamic languages (e.g., JavaScript and Jolie), which are largely employed in the development of microservices [10, 140, 24].

Behavioural Specifications and Choreographies Having formally-defined interfaces in the form of an API is not enough to guarantee the compatibility of services. This is because, during execution, services may engage in sessions during which they perform message exchanges in a precise order. If two services engage in a session and start performing incompatible I/O, this can lead to various problems. Examples include: a client sending a message on a stream that was previously closed; deadlocks, when two services expect a message from one another without sending anything; or, a client trying to access an operation that is offered by a server only after a successful distributed authentication protocol with a third-party is performed.

Behavioural types are types that can describe the behaviour of services and can be used to check that two (or more) services have compatible actions. Session types are a prime example of behavioural types [114, 115]. Session types have been successfully applied to many contexts already, ranging from parallel to distributed computing. However, no behavioural type theory is widely adopted in practice yet. This is mainly because behavioural types restrict the kind of behaviours that programmers can write for services, limiting their applicability. An important example of a feature with space for improvement is non-determinism. In many interesting protocols, like those for distributed agreement, execution is non-deterministic and depending on what happens at runtime, the participants have to react differently [156].

Behavioural interfaces are a hot topic right now and will likely play an important role in the future of microservices. We envision that they will also be useful for the development of automatic testing frameworks that check the communication behaviour of services.

Choreographies Choreographies are high-level descriptions of the communications that we want to happen in a system in contrast with the typical methodology of defining the behaviour of each service separately. Choreographies are used in some models for behavioural interfaces, but they actually originate from efforts at the W3C of defining a language that describes the global behaviour of service systems [105]. Over the past decade, choreographies have been investigated for supporting a new programming paradigm called Choreographic Programming [143]. In Choreographic Programming, the programmer uses choreographies to program service systems and then a compiler is used to automatically generate compliant implementations. This yields a correctness-by-construction methodology, guaranteeing important properties such as deadlock-freedom and lack of communication errors [54, 56, 147].

Choreographies may have an important role in the future of microservices, since they shrink the gap between requirements and implementations, making the programmer able to formalise the communications envisioned in the design phase of software. Since the correctness of the compiler from choreographies to distributed implementations is vital in this methodology, formal models are being heavily adopted to develop correct compilation algorithms [96]. However, a formalisation of how transparent mobility of processes from one protocol to the other is still missing. Moreover, it is still unclear how choreographies can be combined with flexible

deployment models where nodes may be replicated or fail at runtime. An initial investigation on the latter is given in [129]. Also, choreographies are still somewhat limited in expressing non-deterministic behaviour, just like behavioural types.

Moving Fast with Solid Foundations Behavioural types, choreographies, refinement types [178] and other models address the problem of specifying, verifying, and synthesising communication behaviours. However, there is still much to be discovered and developed on these topics. It is then natural to ask: do we really need to start these investigations from scratch? Or, can we hope to reuse results and structures from other well-established models in Computer Science?

A recent line of work suggests that a positive answer can be found by connecting behavioural types and choreographies to well-known logical models. A prominent example is a Curry-Howard correspondence between session types and the process model of π-calculus, given in [53] (linear logical propositions correspond to session types, and communications to proof normalization in linear logic). This result has propelled many other results, among which: a logical reconstruction of behavioural types in classical linear logic that supports parametric polymorphism [187]; type theories for integrating higher-order process models with functional computation [181]; initial ideas for algorithms for extracting choreographies from separate service programs [57]; a logical characterisation of choreography-based behavioural types [58]; and, explanations of how interactions among multiple services (multiparty sessions) are related to well-known techniques for logical reasoning [55, 52].

Another principle that we can use for the evolution of choreographic models is the established notion of computation. The minimal set of language features to achieve Turing completeness in choreographies is known [71]. More relevant in practice, this model was used to develop a methodology of procedural programming for choreographies, allowing for the writing of correct-by-construction implementations of divide-and-conquer distributed algorithms [72].

We can then conclude that formal methods based on well-known techniques seem to be a promising starting point for tackling the issue of writing correct microservice systems. This starting point gives us solid footing for exploring the more focused disciplines that we will need in the future, addressing problems like the description of coordination patterns among

services. We envision that these patterns will benefit from the rich set of features that formal languages and process models have to offer, such as expressive type theories and logics. It is still unclear, however, how exactly these disciplines can be extended to naturally capture the practical scenarios that we encounter in microservices. We believe that empirically investigating microservice programming will be beneficial in finding precise research directions in this regard.

2.4.2 Trust and Security

The microservices paradigm poses a number of trust and security challenges. These issues are certainly not new, as they apply to SOA and in general to distributed computing, but they become even more challenging in the context of microservices. In this section, we aim to discuss some of these key security issues.

Greater Surface Attack Area In monolithic architectures, application processes communicate via internal data structures or internal communication (for instance, socket or RMI). The attack surface is usually also constrained to a single OS. On the contrary, the microservices paradigm is characterised by applications that are broken down into services that interact with each other through APIs exposed to the network. APIs are independent of machine architectures and even programming languages. As a result, they are exposed to more potential attacks than traditional subroutines or functionalities of a large application, which only interacted with other parts of the same application. Moreover, application internals (the microservices) have now become accessible from the external world. Rephrasing, this means that microservices can in principle send the attack surface of a given application through the roof.

Network Complexity The microservices vision, based on the creation of many small independent applications interacting with each other, can result in complex network activity. This network complexity can significantly increase the difficulty in enforcing the security of the overall microservices-based application. Indeed, when a real-world application is decomposed, it can easily create hundreds of microservices, as seen in the architecture overview of Hailo, an

online cab reservation application.[1] Such an intrinsic complexity determines an ever-increasing difficulty in debugging, monitoring, auditing, and forensic analysis of the entire application. Attackers could exploit this complexity to launch attacks against applications.

Trust Microservices, at least in this early stage of development, are often designed to completely trust each other. Considering a microservice trustworthy represents an extremely strong assumption in the "connectivity era", where microservices can interact with each other in a heterogeneous and open way. An individual microservice may be attacked and controlled by a malicious adversary, compromising not only the single microservice but, more drastically, bringing down the entire application. As an illustrative real world example, a subdomain of Netflix was recently compromised, and from that domain, an adversary can serve any content in the context of it. In addition, since Netflix allowed all users' cookies to be accessed from any subdomain, a malicious individual controlling a subdomain was able to tamper with authenticated Netflix subscribers and their data [175]. Future microservices platforms need mechanisms to monitor and enforce the connections among microservices to confine the trust placed on individual microservices, limiting the potential damage if any microservice gets compromised.

Heterogeneity The microservices paradigm brings heterogeneity (of distributed systems) to its maximum expression. Indeed, a microservices-based system can be characterised by: a large number of autonomous entities that are not necessarily known in advance (again, trust issue); a large number of different administrative security domains, creating competition amongst providers of different services; a large number of interactions across different domains (through APIs); no common security infrastructure (different "Trusted Computing Base"); and last but not least, no global system to enforce rules.

The research community is still far from adequately addressing the aforementioned security issues. Some recent works, like [175], show that some preliminary contribution is taking place. However, the challenge of building secure and trustworthy microservices-based systems is still more than open.

[1] hailoapp.com

2.5 Conclusions

The microservice architecture is a style that has been increasingly gaining popularity in the last few years, both in academia and in the industrial world. In particular, the shift towards microservices is a sensitive matter for a number of companies involved in a major refactoring of their back-end systems [81]. This happens not without the help of attractive perspectives microservices offer to systems embracing this architecture. The promises of microservices architecture include the smooth conformity to the ever-changing business requirements, the possibility of components isolation and distribution, and the power of seamless evolution of the systems build with this paradigm.

2.5.1 Challenges and fulfilments

However, despite on all the benefits that the adaptation of microservices architecture could bring, it also has a numerous amount of challenges, some of which were already solved, and some are still remaining open.

Microservices ecosystem

The adoption of any new technology requires an ecosystem to be built around. Between the time of microservices emergence and now, many ad-hoc tools and specialized programming languages have appeared (e.g. [14, 4, 5, 158, 16, 121]). Currently, some research even observes the technology overhead around microservices [151] partly triggered by the buzz of the topic.

Security

Microservices improved scalability puts the security at risk due to the problems mentioned in 2.4.2 (e.g. the increased attack surface and the complexity of the architecture or unjustified trust between services leading to the absence of proper authentication and authorization). As for the other systems, there is no silver bullet, and securing microservices architecture requires thoughtful application of general practices and patterns (e.g. filtering data access with CQRS

pattern [8]) as well as the ones recommended for services-based architectures (e.g. using access tokens and API gateway pattern [9] to focus the problem of authentication and abstract it away from the services behind a gateway). Also, many microservices-centric tools provide out-of-the-box features and solutions for highly-scalable access-control and permissions management, managing secrets and protect vulnerable data [3, 21, 15, 23, 13].

Interfaces and communication correctness

One of the most important challenges of microservices architecture is the maintenance of correct interaction between independent services. To achieve that each service should have a well-defined interface at the local level as well as the whole architecture should possess a global model capable of capturing communication issues, like livelocks, deadlocks or starvation. The importance of solving this challenge is recognized both in academia and industry. In academia, formal methods are actively developing in the direction of expressive behavioral interfaces [26, 116] and models aimed at ensuring the correct communication between services [94, 75], e.g choreographies [96]. In industry, it led to the appearance of a new type of testing applied to microservices architecture - the contract testing [7, 66, 18]. However, there is still a gap between practices used in academia and industry, and adopting the research results in practice still remains an open challenge.

2.5.2 Epilog

Despite the fact that some authors present the microservices architectural style from a revolutionary perspective, we have preferred to provide an evolutionary presentation to help the reader understand the main motivations that lead to the distinguishing characteristics of microservices and relate to well-established paradigms such as OO and SOA. With microservice architecture being very recent, we have not found a sufficiently comprehensive collection of literature in the field, so that we felt the need to provide a starting point for newcomers to the discipline, and offer the authors' viewpoint on the topic.

In this chapter, we have presented a (necessarily incomplete) overview of software architecture, mostly providing the reader with references to the literature, and guiding him/her in our itinerary towards the advent of services and microservices. A specific arc has been given to the narrative,

which necessarily emphasises some connections and some literature, and it is possibly too severe with other sources (for example, with research contributions in the domain of the actor model [111] and software agents [153]). This calls for a broader survey investigating relationships along this line.

Chapter 3

Choreography extraction

Abstract

Choreography is a model of describing a communication behavior in distributed systems, that has a global view on the collaboration of its components. The process of generating choreography from the set of local specifications is called choreography extraction. This work is dedicated to the implementation and the analysis of the choreography extraction algorithm defined by Cruz-Filipe et al. During the implementation, we introduced few optimizations to the algorithm and performed thorough testing, showing the practical application of the algorithm.

3.1 Introduction

This chapter aims to show the results of implementation and analysis of the extraction algorithm for choreographies, presented in the paper [73].

Choreographic Programming [143] is a model for describing communications between elements in distributed systems from the global point of view. Its application includes communication protocols (or, in a more narrow scope, security protocols), distributed algorithms among others. One of the practical benefits of working with choreographies is EndPoint Projection (EPP), an algorithm that translates the global choreographic specification to multiple specifications of local endpoints. EPP guarantees that each local endpoint respects the global specification, therefore the code written by the local specification is correct-by-construction.

However, besides following the top-down approach (projecting from a global choreography to local specifications), a bottom-up one might be required. One of the possible cases is when local specification changes. Making it unilateral deprives the local specification of the global specification conformity and, thus, makes it potentially incorrect. The algorithm of deducting the choreography by the set of local endpoints specifications is called choreography extraction.

The extraction algorithm is based on an abstract execution graph built by all possibles derivations of the choreography. It works both in synchronous and asynchronous settings, allowing to process the specifications that would deadlock the choreography in synchronous settings. This algorithm has been proven to have exponential execution complexity. This lower complexity results from not allowing processes to receive messages from non-deterministic sources.

While implementing the extraction algorithm, we have decided to introduce some changes. Subsection 3.3.3 shows the algorithm details, and section 3.4 shows the changes.

3.2 Related work

Choreography models and application

Choreography models is a vibrant research topic that is presented extensively in the literature [54, 51, 45, 44, 36]. Choreography models are mostly equipped with an algorithm for projecting endpoints [163, 112, 106, 126, 73]. [112] uses the Dynamic Condition Response graphs to express choreographies and end-points, [106] introduces an abstract semantics framework for choreographies (in the form of global graphs) that can be projected to communicating finite state machines.

Choreography theory is related to the theory of multiparty session types, which is also used for describing communication protocols and checking if local implementations respect them [115, 60, 125, 39, 62].

Choreographies has impacted industry standards, languages, and ways of building software [65, 42, 20, 49, 29]. Scribble [171, 193] is a programming language based on the theory of multi-party session types, using to describe protocols for communicating systems. StMungo [123] is a tool providing type checking for the local endpoints projected with Scribble. Chor [64, 143] is the first implementation of a programming language based on choreographies. AIOCJ [75, 93] is a dynamic choreographic language, which supports on the fly adaptation of running applications. DIOC (Dynamic InteractionOriented Choreography) [76] is a choreographic programming language allowing the programmer to specify the parts of the application to be updated. The language presented in [59] exploits both the theory of session types and choreographies to be applied for reactive programming.

Choreography extraction

To the best of authors knowledge, there exist only a few lines of research on choreography models supporting both top-down and bottom-up approaches.

[126, 33] continue the work presented in [125]. Bartoletti et al. study the use of choreographies in an untrustworthy environment where participants may be violating their expected behavior, and present the model, where each participant ought to present its expected behavior as a

contract [33]. Here services may be composed according to their local session types to create a choreography as a global type if their contracts admit this agreement. Lange et al. presents an algorithm building graphical choreographies from communicating finite state machines (CFSMs) embodying asynchronous interactions between choreography components [126]. This extraction algorithm here uses CFSMs as local specifications. The algorithm requires all process moves and choices to be presented in the label transition system, which has a negative impact on the algorithm efficiency (super-factorial time, the worst case).

In [73] Cruz-Filipe et al. continue the work started in [57] and present the extraction algorithm supporting asynchronous communications, which shows efficient treating of such choreographies (exponential time, the worst case).

3.3 Background

3.3.1 Model behind the extraction

This subsection revises the formal languages behind the extraction algorithm: the language of Core Choreographies (CC) and the language of Stateful Processes (SP) [73].

The syntax of CC is given in Figure 3.1.

$$C ::= 0 \mid \eta; C \mid \text{if } p \overset{\leq}{=} q \text{ then } C_1 \text{ else } C_2 \mid \text{def } X = C_2 \text{ in } C_1 \mid X$$
$$\eta ::= p.e \text{ -> } q \mid p \text{ -> } q[l] \qquad\qquad e ::= v \mid * \mid \ldots$$

Figure 3.1: *Core Choreographies, Syntax.*

A choreography body C can be one of the following terms:

- 0, the terminated choreography;
- $\eta; C'$, an interaction η followed by the choreography C';
- $\text{if } p \overset{\leq}{=} q \text{ then } C_1 \text{ else } C_2$, a conditional; if a value of a process p equals to a value of a process q, execution continues as C_1, otherwise as C_2;

– X, a procedure defined by the term $\mathsf{def}\, X = C_2 \,\mathsf{in}\, C_1$, that can be called in C_1 and C_2

An interaction term η can be a communication $p.e$ -> q between processes (process p sends value e to process q) or a selection p -> $q[l]$ (process p selects a branch labeled with l among the branches offered by q). We assume expression e to be of sort types.

The syntax of SP is given in Figure 3.2. A network N either is terminated (denoted as 0),

$$B ::= q!\langle e\rangle;B \mid p?;B \mid q \oplus l;B \mid p\&\{l_i : B_i\}_{i\in I} \mid \quad N ::= p \triangleright B \mid 0 \mid N \mid N$$
$$\mid 0 \mid \mathsf{if} * \overset{\leftarrow}{=} q \,\mathsf{then}\, B_1 \,\mathsf{else}\, B_2 \mid \mathsf{def}\, X = B_2 \,\mathsf{in}\, B_1 \mid X$$

Figure 3.2: *Stateful Processes, Syntax.*

or it contains a process p with a behavior B. Processes in a network behave as the following terms:

– $q!\langle e\rangle;B$, sending: process p sends the value e to the process q and then continues as B;

– $p?;B$, receiving: process q receives a value from p and then continues as B;

– $q \oplus l;B$, selection: process p sends the selection of the branch labeled l to the process q;

– $p\&\{l_i : B_i\}_{i\in I}$, offering: process q offers a choice among the labels l_i to the process p and runs the behaviour corresponding to the chosen label;

– $\mathsf{if} \overset{\leftarrow}{=} * q \,\mathsf{then}\, B_1 \,\mathsf{else}\, B_2$, conditional: checks if the value stored at the process q is the same as the value stored at the process which runs this behaviour, and proceeds as continuations B_1 or B_2 correspondingly;

– X, a procedure with body C_2 defined as $\mathsf{def}\, X = B_2 \,\mathsf{in}\, B_1$, that can be called in C_1 and C_2.

Sending and selection are called output terms. Dual to them are input terms (receiving or offering correspondingly)[1].

[1]NB: It does not mean that a process must have a dual term in a network. Consider the example inspired by *Snow White and the Seven Dwarfs* fairy tale[103]: Seven Dwarfs saw that Snow White had been there because everything was not in the same order in which they had left it, and each of them asked: "who has been doing it." This behavior is captured by the well-formed network: `dwarf1|> if sitOnMyChair then dwarf2!<"Who?">;0 else 0` in which `dwarf2!<"Who?">` action does not have a dual one.

3.3.2 Changes to the model

During the implementation of the extraction algorithm, we have slightly changed the underlining models.

Figure 3.3 shows the revised syntax of the CC. The root element in our syntax is not a choreography, but a program P, containing a pair of a main choreography C and $\mathscr{D}$, the set of procedure definitions. Instead of having a constructor $\mathsf{def}\, X = C_2 \,\mathsf{in}\, C_1$ for each recursive definition in a choreography, we treat them all as pairs of a procedure name and a choreography constituting the procedure body (including the initial choreography C itself as a procedure with name C and its body as a procedure body).

Instead of using $\mathsf{if}\, p \overset{\leftarrow}{=} q \,\mathsf{then}\, C_1 \,\mathsf{else}\, C_2$ (p checks if its value is equal to the q's and proceeds either as C_1 or as C_2) in the syntax of CC we use the local conditional $\mathsf{if}\, p.e \,\mathsf{then}\, C_1 \,\mathsf{else}\, C_2$ proposed as improvement in [73]. This construct is more akin to mainstream programming languages, which use local conditionals.

$$
\begin{aligned}
P &::= < \mathscr{D}, C > \\
\mathscr{D} &::= X\{C\}, \mathscr{D} \mid 0 \\
C &::= 0 \mid \eta; C \mid \mathsf{if}\, p.e \,\mathsf{then}\, C_1 \,\mathsf{else}\, C_2 \mid X \\
\eta &::= p.e \text{ -> } q \mid p \text{ -> } q[l]
\end{aligned}
$$

Figure 3.3: *Core Choreographies, Revised syntax*

Figure 3.4 shows the revised syntax of the SP. Each network N either is terminated, or it contains a process p with consisting of a pair of a main behavior B and $\mathscr{D}$, the set of procedure definitions. To express the parallel composition of processes, we use the term N|N.

SP treats the procedure definition and condition terms similar to what CC does. We use the set of procedure definitions $\mathscr{D}$, consisting of pairs of a procedure name X with some behaviour B, and we use local condition $\mathsf{if}\, e \,\mathsf{then}\, B_1 \,\mathsf{else}\, B_2$ instead of the one proposed in [73].

Example 3.3.1. Let us accompany the formal definition of CC and SP with a practical example of the choreography and the network of a simplified version of the TCP protocol [161]. We have omitted actions happening at timeout and assumed that all messages being sent are always

$$
\begin{aligned}
N &::= p\triangleright < \mathscr{D}, B > \mid 0 \mid N \mid N \\
\mathscr{D} &::= X\{B\}, \mathscr{D} \mid 0 \\
B &::= q!\langle e\rangle; B \mid p?; B \mid q \oplus l; B \mid p\&\{l_i : B_i\}_{i\in I} \mid \text{if } e \text{ then } B_1 \text{ else } B_2 \mid X \mid 0
\end{aligned}
$$

Figure 3.4: *Stateful Processes, Revised syntax.*

received. The TCP algorithm comprises three parts: handshake, actual transmission, and termination described below in more detail.

1. Handshake
   (a) Client sends *syn* token to server.
   (b) Server receives *syn* and tries to create a socket for client. If it succeeds, it sends *syn* and *ack* tokens to client. If it fails, it sends *rst* token to client.
   (c) If client receives *rst*, it does not try to connect further. If client receives *syn* from server, it sends back *ack*.
   (d) When both server and client receive *syn* and *ack* tokens, they enter *Established* state.
2. Transmission
3. Termination
   (a) If client wants to terminate transmission, it sends *fin* token to server.
   (b) Server sends back *fin* and *ack* tokens as confirmation of transition termination.
   (c) Client sends *ack* token to server as confirmation from its side.

The choreography below expresses this algorithm. We have left the content of transmission phase empty, as it is of no importance for the current example.

```
1  def main { TCP }
2  def TCP {
3    client.syn -> server;
4    if server.createSocketSuccess
5        then
6            server -> client[synRcv];
```

```
7             server.syn -> client;
8             server.ack -> client;
9             client.ack -> server;
10            Established
11        else
12            server -> client[rst]; 0
13  }
14  def Established {
15    if client.transmissionInProgress
16        then
17            client -> server[transmission];
18            ...
19            Established
20        else
21            client -> server[fin];
22            server.fin -> client;
23            server.ack -> client;
24            client.ack -> server;
25            0
26  }
```

The choreography comprises of two recursive procedures: the main procedure *TCP*, used for handshaking communication; and the *Established* procedure, where transmission takes place and where the termination procedure initiates.

The choreography description is reasonably verbose. However, we would like to point out few details.

Lines 3–5 and 22–23 of the choreography implements step 1.b of the algorithm. Here, instead of basic communication, interaction with guarded choice was used to pass the *syn* or *rst* token from server to client to divide further behaviours in branches.

To simulate the actual transmission, for each transmission action in lines 10–14, we check if the transmission is still in progress, send the *transmission* token to server, do some actions, and recursively call *Established* again.

The following figure shows the corresponding network, obtained by the EPP algorithm.[2]

```
  server |>
    main { TCP }
    def TCP {
        client?;
        if createSocketSuccess
            then
                client + synRcv; client!<syn>;
                client!<ack>; client?; Established
            else
                client + rst; 0
    }
    def Established {
        client&{
            transmission: Established,
            fin:
                client!<fin>; client!<ack>; client?; 0
        }
    }

| client |>
    main { TCP }

    def TCP {
        server!<syn>;
        server&{
            rst: 0,
            synRcv: server?; server?; server!<ack>; Established
        }
    }

    def Established {
        if transmissionInProgress
            then
                server + transmission;
```

[2]EPP details are out of scope of this chapter and can be found in [73]

```
35                  ...
36                  Established
37              else
38                  server + fin; server?; server?;
39                  server!<ack>; 0
40      }
```

Here, we have two processes, server and client, with parallel behaviours. Each process also has two recursive procedures *TCP* and *Established*, each action of which is one half of input/output pair, composing together a corresponding action from the choreography above.

3.3.3 Extraction algorithm

We consider the cases of networks of synchronous and asynchronous settings separately, and start with the synchronous case as the simplest of these two.

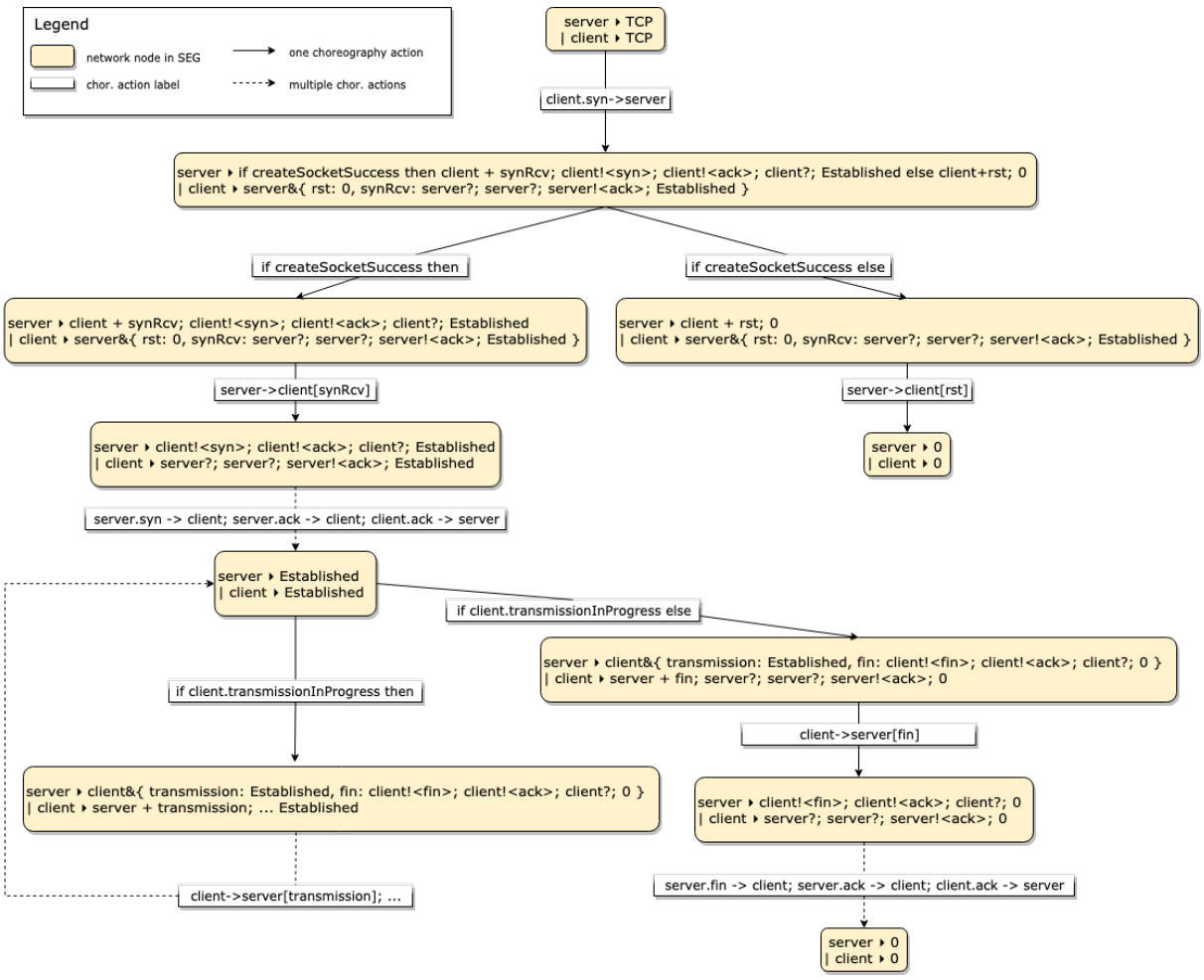
Synchronous case

The extraction algorithm works on a Symbolic Execution Graph (SEG). A SEG is a directed graph and a subgraph of a more global entity called Abstract Execution Space (AES). The AES for a network N depicts all possible abstract reduction paths from N. Each node in AES contains the network resulting from a reduction step $\mathsf{N} \xrightarrow{\tilde{\alpha}} \mathsf{N}'$, with the edge between the nodes labeled as corresponding α. The AES represents all possible evolutions of a network, while the SEG contains a fixed and ordered sequence of actions in the network, abstracted from the state. Following the semantics rules presented in [73], each node in SEG can have only one or two outgoing edges representing labeled reduction with labels from CC: in the first case, it is labeled as η, in the second, as p.e then and p.e else.

In case if there are no matching actions in the network, the construction of the SEG can not be finished. We can deal with it in several ways. We can restrict the extraction to the lock-free networks only, or we can extract the stuck processes to a new choreography term **1**. Implementing the algorithm, we went for the first option, raising an exception, if we identify a network that can not be extracted because of a deadlock.

Building SEGs for finite networks, not containing recursive definitions, is rather trivial since, in this case, there is no risk that the AES is infinite. Treating recursive definitions, on the other hand, needs to be approached more carefully due to infinite recursive unfolding breaking the guarantee that extraction eventually terminates. One way to avoid an infinite unfolding is to restrict it to be performed only once: when met on the top of the process and when involved in a process reduction. Figure 3.5 shows the SEG for TCP algorithm presented earlier.

Figure 3.5: *SEG of TCP algorithm*



Another problematic spot is the possible starvation of a process. Imagine a network N of several processes, one of which having a call to a recursive procedure X on top. If we reduce actions of this process only, the network reduction in SEG may evolve at some point to the term from which reduction process was started, and extracting from such a SEG will yield a choreography without any trace of processes that did not participate in X. To deal with this, for each $N \xrightarrow{\tilde{\alpha}} N'$ we mark in N' all procedure calls that were unfolded and participated in α. If all procedure calls

in the network are marked, we erase all the marking. While building the SEG we require the fulfilment of two conditions: all loops shall contain a node with all procedure calls unmarked, meaning that all procedures were unfolded at least once before going to the original node; and all loops shall start having all processes with finite behaviour deadlocked (or terminated).

Example 3.3.2. Let us imagine a system of two communicating pairs of processes. Each process greets its match, wait for its reply, and then the process repeats.

```
  a |> main {Greet} def Greet{b!<hi>; b?; Greet}
| b |> main {Greet} def Greet{a?; a!<hi>; Greet}
| c |> main {Greet} def Greet{d!<hi>; d?; Greet}
| d |> main {Greet} def Greet{c?; c!<hi>; Greet}
```

Without implementing the marking system, the extracted choreography has one of two possible forms. In each case, the presence of the second pair is ignored.

```
I: def Greet{a.hi->b; b.hi->a; Greet}
II: def Greet{c.hi->d; d.hi->c; Greet}
```

Figure 3.6 shows two corresponding SEGs. Without marking, after each of two possible reductions, the network evolves to the root node.

Figure 3.6: *Greeting example SEGs (without marking)*

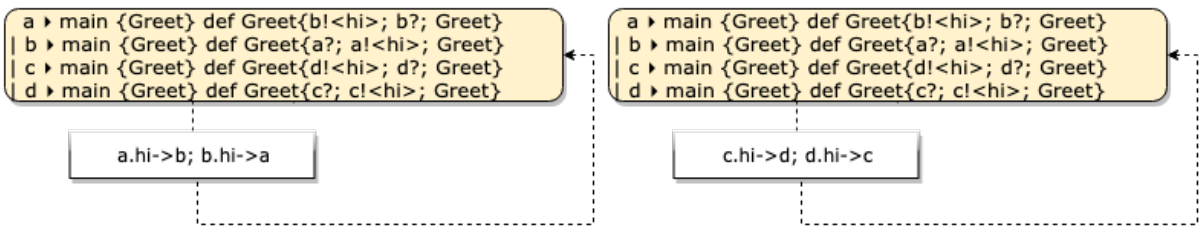
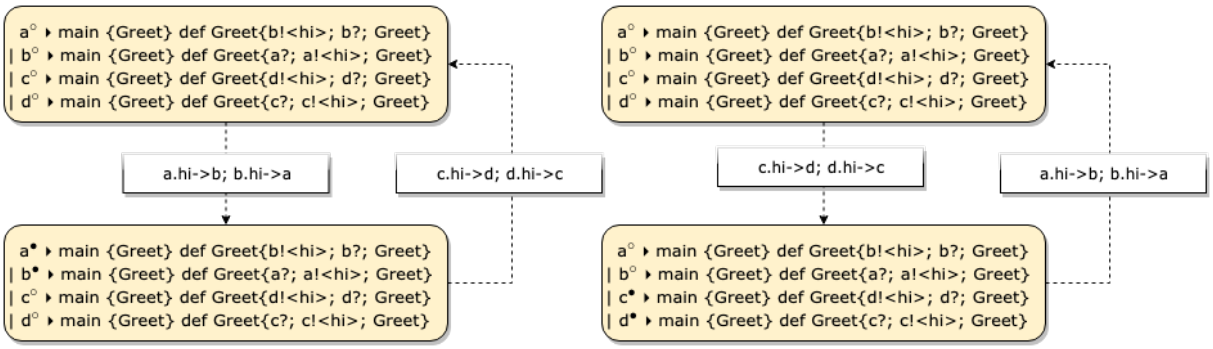


Figure 3.7 shows two SEGs built with marking. Here in both cases, after the first reduction, the SEG evolves to a network different from the root, which forces it to make the second reduction. These SEGs produce the correct choreographies, which differ only by order of reductions.

```
I: def Greet{a.hi->b; b.hi->a; c.hi->d; d.hi->c; Greet}
II: def Greet{c.hi->d; d.hi->c; a.hi->b; b.hi->a;Greet}
```

The idea behind the extraction of recursive definitions from SEG is to to extract them from loops in the SEG rather than from the definitions from the source network [73]. In order to do that, we

Figure 3.7: *Greeting example SEGs (with marking)*



identify all nodes that have more than one incoming edge (which shows the presence of a loop). Then, for each of these nodes, we remove all incoming edges to the node and direct them to a new node of a special type, that contains the name of procedure being called. In this way the graph becomes a set of trees with root either a N either a recursive definition. fig. 3.8 shows the result of loop elimination of the SEG from fig. 3.5.

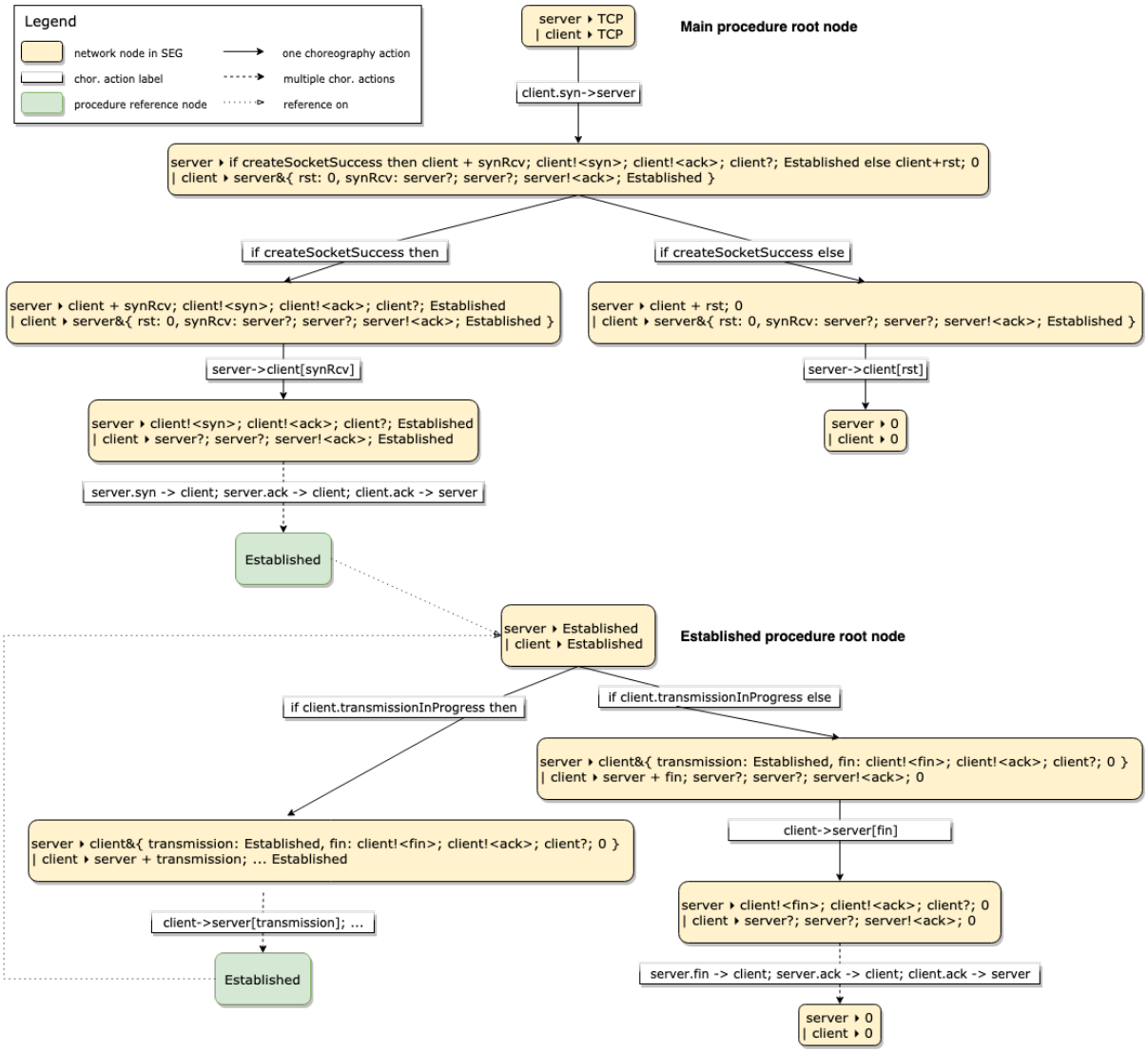
Now we can extract the choreography starting from the initial network N and then extracting all recursive definitions. It starts with the node labeled with N and checks the outgoing edges. If there is one edge and from N to some N' and it is labeled as η, we add η to the choreography and continue with N'. If there are two outgoing edges from N: $N \xrightarrow{p.e:\text{then}} N'$ and $N \xrightarrow{p.e:\text{else}} N''$, we write to choreography a conditional with then and else branches as a result of extracting the N' and N'' correspondingly.

Asynchronous case

Dealing with asynchronous case requires some changes in the language of CC. We introduce a new choreographic action, called multicom, which is a list of communication actions with distinct receivers [73]. We also change the definition of AES (and SEG) labels, so now they can be of a multicom kind.

For its part, the extraction algorithm is extended by asynchronous behaviour with synchronisation on queues for each pair of asynchronous processes (e.g sending actions writes messages to the receiver's queue). In order to extract a multicom from a network we use the following algorithm:

Figure 3.8: *TCP SEG loop elimination*



1. For each process in the network with sending or selection behaviour, we put the corresponding choreographic action to the receiver's queue (called waiting).

2. Until we empty the queue, we move an action from it to the list of actions. If the behaviour of this action's receiver consists only of sending or label selection actions before a next receiving action, we put all these actions to the receiver's queue (if they are not in the actions list already).

3. We return back the list of actions that will constitute a multicom.

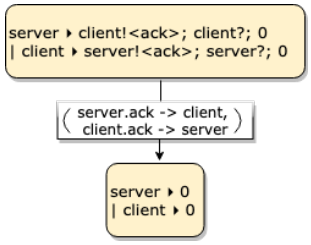
As long as the SEG is built, extraction algorithm flows exactly as in the synchronous case.

Example 3.3.3. To confirm the transaction termination in TCP protocol, a server and a client need to exchange *ack* tokens. This might be done synchronously or asynchronously, like in the following example.

```
1   server |> client!<ack>; client?; 0
2 | client |> server!<ack>; server?; 0
```

The SEG for the asynchronous case is shown on the fig. 3.9

Figure 3.9: *SEG with multicom action*



3.4 Changes in extraction algorithm

Bringing theoretical ideas to practice can sometimes motivate for changes in algorithm. Working on implementation of extraction algorithm, we have decided to make some amendments.

3.4.1 Treating loops

Most changes of the algorithm we have introduced regards treating the loops in SEG. In order to be able to identify the loop correctly we have started to use node prefixes, lists of already encountered nodes and have rethought marking of the procedures.

Marking

Instead of marking the procedure definition being unfolded and called, we mark the processes containing it. From the point of view of the algorithm, it does not change anything, because we are interested in knowing that a process has been participated in the communication and not

which exact procedure was called, but from the point of view of implementation it is easier to check if the process is marked, than checking all its procedures.

When we acquire a node, where all processes were visited, we erase the node's marking, but we mark the label between this node and its predecessor, so we can easily spot it later.

Bad nodes

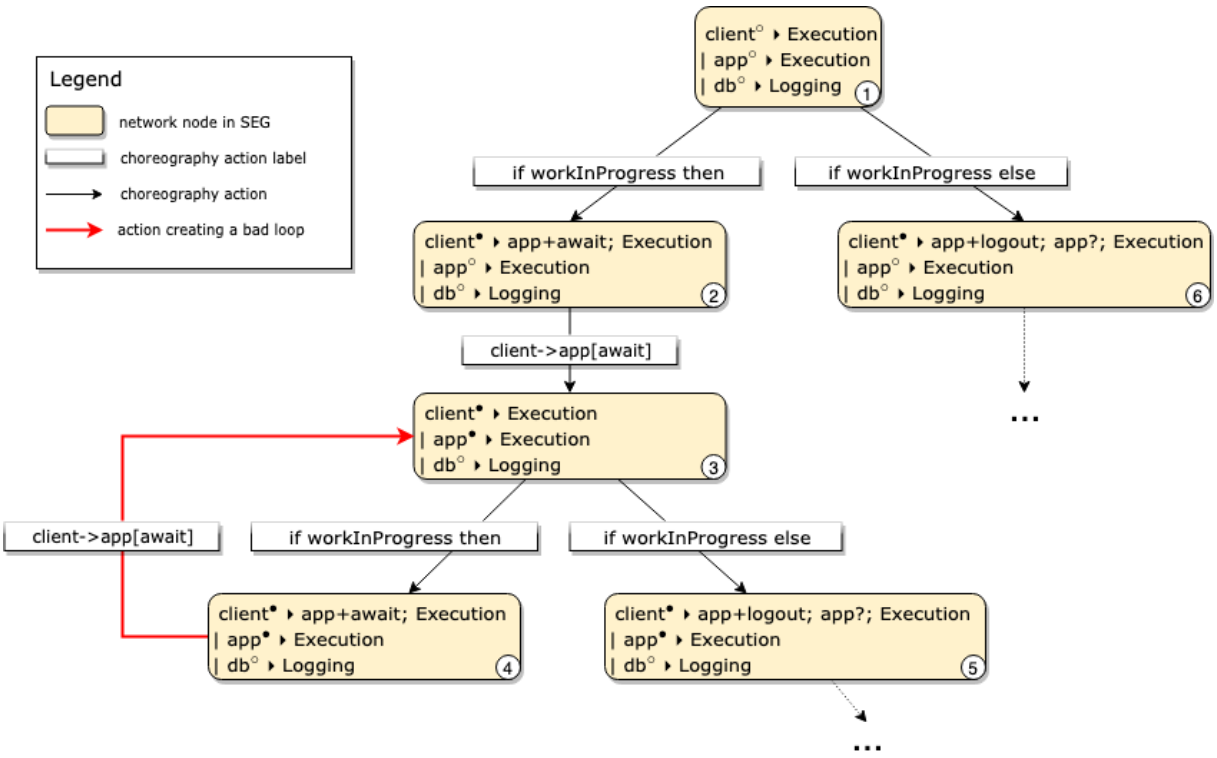
Closing a loop in the SEG we want to know whether all processes in the nodes of this loop were visited before. If a closing label between the source and the target nodes is flipped, it would give us an immediate answer. In other cases we would need to trace back all incoming labels of all descendant nodes. In order to avoid that, each node stores the list of its descendants from the point when the label was flipped the last time. If any node wants to close the loop with a node which its descendant, that clearly indicates that not all processes in this node were visited, and so this is a unsound loop.

Example 3.4.1. Let's look at the network, describing the interaction between a client, an application, and a database. While client is working with app, it repeatedly sends *await* token to it, and when he finishes the work, he sends *logout* token. In this case, app sends logs to db and notifies client about it.

```
client |>
    def Execution{
        if workInProgress
            then app+await; Execution
            else app+logout; app?; Execution}
    main {Execution}
| app |>
    def Execution{client&{
            await: Execution,
            logout: db!<log>; client!<syncLog>; Execution}}
    main {Execution}
| db |>
    def Logging{app?; Logging}
    main {Logging}
```

The SEG for this network can not be built, because it has a bad loop between the third and the fourth nodes: the label between these nodes was not flipped (because db process has not been visited, and so the marking wasn't erased) and the fourth node has a third one in the list of its descendants.

Figure 3.10: *SEG of the network with livelocked process (presence of a bad loop)*



Network Prefixes

Sometimes a loop might be unsound even if it satisfies the conditions states above. We have introduced prefixes for the nodes, to guard the conditional branches and disallow creating an unsound loop e.g. if N' is equal to N''. This amendment is caused by the fact that we traverse the graph depth-first. It means that, for example, in case of conditionals we fully build one branch, and so the second one might try to connect to the node belonging to the first one, creating an unsound loop.

The first node in the SEG with the network N has the prefix **0**. If the next reduction is $\mathsf{N} \xrightarrow{\tilde{\eta}} \mathsf{N}'$, the N' obtains the same prefix as N has; but if the reductions are $\mathsf{N} \xrightarrow{p.e:then} \mathsf{N}'$ and $\mathsf{N} \xrightarrow{p.e:else} \mathsf{N}''$, the N' will have the prefix equal to **00** and the N'' will have **01**.

Example 3.4.2. SEG presented on the fig. 3.5 has two nodes with the same network:

```
  client |> 0
| server |> 0
```

It happens, because these nodes have different choice paths, the node above has the choice path equals to **01**, and the node below - **001**

3.4.2 Livelocked processes

For some networks presence of livelocked processes is normal. However, the extraction algorithm will fail on them, because they will not pass the check on presence of bad loops (they will be not participating in any communication and thus not marked). To make their extraction possible, we mark the livelocked processes from the very beginning and do not erase their marking.

Example 3.4.3. To fix SEG for example 3.4.1, we permanently mark db process. In this case after the execution of the action between the second and the first process, the label will be flipped, because the unmarked db process will not impede in marking erasing . Figure 3.11 shows the fixed SEG.

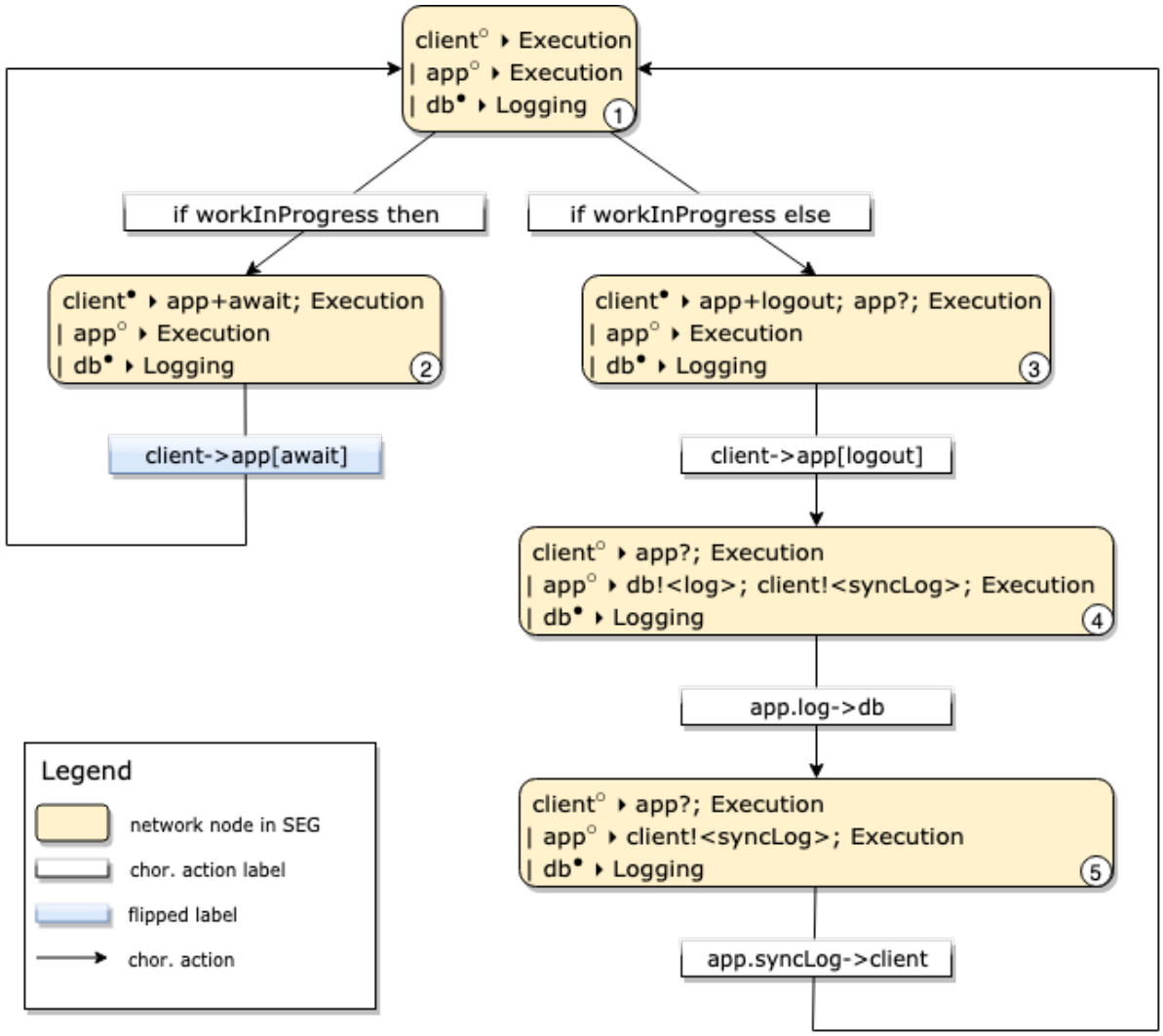
3.5 Implementation

This section explains implementation details of the extraction algorithm.

As it was described previously, we use a directed graph to represent possible network evolutions. We start building the graph with a node, that contains the network to be extracted. Each subsequent node represents a new network state after the execution of an action in the network. The information about the execution taken place is stored in the edge between the involved two nodes; later it will be used for building a choreography. There might be one or two outgoing edges from a node: one is for actions like sending, selection and procedure invocation, and two is for the conditional branches.

If the network contains recursive procedures, the graph will have cycles. We extract those as a procedure definition. As mentioned earlier, to traverse such we flatten all loops by breaking the

Figure 3.11: *SEG of the network with livelocked process (fixed)*



edge between the node with the procedure call and the subsequent one, which itself becomes a root node for building the procedure's body.

Eventually, we might divide our extraction algorithm into three parts

1. Building the execution graph from the network (See 3.5.3)
2. Removing loops (See 3.5.4)
3. Assembling the choreography by collecting its actions from the edges in the graph (See 3.5.5)

All required settings for the algorithm are done in the functions *extractChoreography* and *extract* described in 3.5.2, there are also several auxiliary functions involved in the algorithm and mentioned in the pseudocode. They can be found in the section 3.5.6.

To deal with problem of networks with starving processes, that appear only in networks with recursive procedures, we want to mark the processes whose term has been already executed, to avoid its infinite execution. For these purposes each node in the graph stores the network's marking. We mark as visited processes that have no active actions in the main procedure or which are in the list of the processes that we allow to be livelocked (we have to provide the list with such processes, if any).

Apart from storing the network and marking, the datatype of nodes has such auxiliary attributes as an unique identifier, the set of identifiers of previously encountered nodes and the choice path.

```
data class Node
  (val network: Network, val choicePath: String, val id: Int, val badNodes:
      Set<Int>, val marking: Map<ProcessName, Boolean>)
```

We store globally hashes and path choices of all nodes, to make searching and comparing of the nodes faster and we also have a counter variable to create nodes identifiers. These variables are mutated implicitly by functions used in the extraction algorithm.

```
val graph: DirectedGraph<Node, ExtractionLabel>()
val nodeHashes = Map<Hash,List<Node>>()
var choicePaths = Map<String,List<Node>>()
var nodeIdCounter = 0
```

3.5.1 Extraction strategies

Extraction strategies define the order of actions execution. For example, apart for choosing the next action randomly, we can take conditional or selection action first, or take the process with shortest term (with least amount of actions). Choosing different extraction strategies may vary the shape of execution graph and the time of extraction.

We devised the following extraction strategies:

1. `InteractionsFirst`. This strategy takes the first process which has sending, receiving, selection or offering action on top;
2. `ConditionsFirst`. Takes the first process with conditional action on top;
3. `Random`. Randomly chooses a process;
4. `UnmarkedFirst`. Takes first process which has not been marked yet;
5. `UnmarkedThenInteractions`. Takes any unmarked process, which has sending or receiving action on top;
6. `UnmarkedThenSelections`. Takes any unmarked process, which has selection or offering action on top;
7. `UnmarkedThenRandom` Takes any unmarked process;
8. `UnmarkedThenConditions`. Takes any unmarked process, which has conditional action on top;
9. `ShortestFirst`. Takes the process with the shortest main procedure.
10. `LongestFirst`. Takes the process with the longest main procedure.

3.5.2 Setting up the extraction

Function *extractChoreography* is the entry point of the extraction algorithm. As parameters it takes the network to be extracted, an extraction strategy and the list of livelocked processes, since processing the network with such processes will lead to a bad loop.

Look at the network of a bargain process between a client, a seller and a bank. The seller offers the client a price, and the client can agree on it or continue bargaining. In the first case, the client selects the *happy* label offered by the seller and sends the bargain details to the bank. In the second case, if it refuses the offer, it selects the *no* label and the seller needs to offer a new price again. If the client is always unhappy about the price, bargaining between the client and the seller can continue infinitely long, which makes the bank process livelocked.

```
1    client {
```

```
        def X { if ok
            then seller+happy; bank!<info>; stop
            else seller+no; seller?; X }
        main {seller?; X} }
|   seller {
        def X {client&{
            no: client!<price>; X,
            happy: stop} }
        main {client!<price>; X} }
|   bank { main {client?; stop} }
```

Before starting the extraction process, we preprocess the network. We clean the network from the processes whose terms were already terminated with the function *purgeNetwork*. If there is no processes with active terms, we prune the network to have only one process (not important which one). With function *splitNetwork* we analyze the interaction between processes and split them to different networks if there is no interaction between the processes in these networks involved. For instance, we can expand the bargain example described above and add processes client2, seller2 and bank2 interacting only with each other. In this case there is no need to threat them all as a homogeneous group and better to divide into two networks, one consisting of client, seller and bank, and another consisting of client2, seller2 and bank2 processes.

We run extraction on each of the parallel networks with the *extract* function. It starts with creating a map of processes marking in the network (marks terminated and livelocked processes). Then it creates a node, based on the input network and marking. The choice path of this node is equal to 0 and the set of bad nodes is empty. It adds the node to the graph, the map of path choices and to the map of hashes.

Next, when everything is set, it tries to build a graph by calling function *buildGraph*. In case of success, it removes loops from the graph with *unrollGraph* and returns the choreography produced by the *buildChoreography* function. Otherwise, a choreography can't be extracted and only *null* is returned.

Finally, *extractChoreography* creates the variable of data structure *Program* that contains the list of all extracted choreographies.

```
fun extractChoreography
```

```
  (n:String,
   strategy:ExtractionStrategy,
   services:List<String>)
  :Program {
    val inputNetwork = purgeNetwork(n)
    val parallelNetworks = splitNetwork(inputNetwork)
    for (network in parallelNetworks){
        results.add(extract(network))
    }
    return Program(results)
}

fun extract
  (n: Network)
  :Choreography {
    val marking = Map<ProcessName,Boolean>()
    for (name,term in n.processes) {
        marking[name] = (term.main is TerminationSP) || services.contains(
            name)
    }
    val node = Node(n,0,nextNodeId(),emptySet,marking)
    graph.addnode(node)
    addToChoicePathMap(node)
    addToHashMap(node)
    if (buildGraph(node)) {
        val procedureNodes = unrollGraph(node)
        return buildChoreography(node,procedureNodes)
    } else
        return null
}
```

3.5.3 Building the graph

Function *buildGraph* builds the execution graph for the network. This is a recursive function which takes a node, strives to build the next one (or next two nodes in case of conditional action) and calls itself on the new node(s), until no possible actions are left in the network. If at this point

all processes were terminated, that means that building the graph was successful, so *buildGraph* returns true, otherwise it returns false.

Before finding an action for execution, *buildGraph* sort the network's processes with respect to the extraction strategy in *copyAndSortProcesses* function. Then it "unfolds" all processes that have procedure invocations on top, meaning that it replaces the procedure invocation on the corresponding procedure term, and stores the information about processes being unfolded in the set of unfolding processes. We decided to unfold all actions before looking for a possible execution action and than fold back the ones that were unused, rather than unfold an action exactly at the place where we meet it, because it appears to be more effective - all non-trivial networks contain multiple procedure invocations. Next it picks one by one the sorted processes and tries them against the presence of a single action interaction: the communication or the conditional one.

Let's have a look at the first one, finding a communication. The *findCommunication* function is in charge of searching for sending/receiving or offering/selection action. If it succeeds, *findCommunication* returns the pair of the new target network, created from the current one by executing the communication action, and the label which stores the information about this action, so now *buildCommunication* can procced with building the communication action. First it folds back all processes that were unfolded and have not been participated in this interaction. Then it tries to find a place for the new network in the graph in the in the *buildCommunication* function, which hides all subsequent steps of building a new node and adding it to the graph. Here, we will know if it was successful only by the value *buildCommunication* returns - if it is true, that means that the graph was built, so the *buildGraph* can now also return true.

Nevertheless, *buildCommunication* can also fail. It might happen for example due to the occurrence of a bad loop in the graph (will be discussed further). In this case the state of the graph will be rolled back to the stable point before the failure, all nodes created after that point will be removed, and *buildGraph* will now strive to find a conditional among the network processes.

Finding a conditional has the similar logic as finding a communication. If there is such action, *findConditional* will return two new target networks and labels. Then in *buildGraph* it folds back all the processes except the one where conditional action tool place. Then it tries to find a place

for the nodes with two new networks in the graph by the means of the *buildConditional* function. If *buildConditional* succeeds *buildGraph* returns true.

If *buildConditional* fails, meaning that there is no single-action interaction left in the network, buildGraph will move forward to check if all processes were already terminated, so it could exit with success. If it is not true, it also needs to check for presence of multi-action interaction, when a group of processes wants to send messages to a group of receivers, with circular dependencies among communications.

To collect the set of such actions we follow the algorithm:

1. For each process in the network with behavior that starts with sending action or selection, we set actions as an empty set and put to the waiting set the first action of the process term.
2. Then, until the set of waiting actions is not empty, we move an action from waiting to actions and check the term of the action's receiver (since we are dealing with sending and selection operations). If the term of the receiver is of the form $a_1;...;a_k;r?;B$ where each a_i is either the sending of a value or a label selection, then: for each a_i, if the corresponding choreography action is not in actions, we add it to waiting (the case for label selection is similar).

The first step of the algorithm is being performed inside *buildGraph*. It looks among the network's processes to find if one of them is an interaction — sending or selection to create a label with the process sender, receiver and expression (see the *createInteractionLabel*). If it manages to do that, it puts the label to the set of waiting actions and pass it among other parameters to the *collectMulticomActions* function, where the second step is performed. If it manages to collect at least two such actions, *buildGraph* calls *buildMulticom*, whose purpose is the similar to the purpose of *buildCommunication* and *buildConditional*. If it succeeds, *buildGraph* imidiately returns true, otherwise it repeats multicom algorithm for the rest of remaining processes.

If no single- or multi-action interaction was found and not all processes were terminated, *buildGraph* fails.

```
1 fun buildGraph
2   (currentNode: Node)
3   : Boolean {
```

```
4       val processes = copyAndSortProcesses(currentNode)
5       val unfoldedProcesses = Set<String>()
6       for ((processName,processTerm) in processes) {
7           if (unfold(processName,processTerm))
8               unfoldedProcesses.add(processName)
9           }
10      }
11
12  // Try to find an interaction or a conditional
13      for ((processName,processTerm) in processes) {
14          val unfoldedProcessesCopy=Set(unfoldedProcesses)
15
16  // Try to find an interaction
17          val communication = findCommunication(processes,processName,
                processTerm)
18          if (communication!=null) {
19              val (targetNetwork,label) = communication
20              unfoldedProcessesCopy.removeAll(label.sender,label.receiver)
21              fold(unfoldedProcessesCopy,targetNetwork,currentNode)
22              if (buildCommunication(targetNetwork,label,currentNode))
23                  return true
24              else
25                  continue
26          }
27
28  // Try to find a conditional
29          val conditional = findConditional(processes,processName,processTerm
                )
30          if (conditional!=null) {
31              val (thenNetwork,thenLabel,elseNetwork,elseLabel) = conditional
32              unfoldedProcessesCopy.remove(thenLabel.process)
33              fold(unfoldedProcessesCopy,thenNetwork,currentNode)
34              fold(unfoldedProcessesCopy,elseNetwork,currentNode)
35              if (buildConditional(thenNetwork,thenLabel,elseNetwork,
                    elseLabel,currentNode))
36                  return true
37              else
```

```
                continue
            }
        }

// Check if there are no possible actions left
        if (allTerminated(processes)) return true

//try to find a multi-action communication
        for (processName in processes) {
            val interactionLabel = createInteractionLabel(processName,processes
                )
            if (interactionLabel != null) {
                val actions = List<InteractionLabel>()
                val waiting = List<InteractionLabel>()
                waiting.add(interactionLabel)
                collectMulticomActions(waiting,actions,processes,currentNode)
                if ((actions.size > 1) && (buildMulticom(actions,currentNode,
                    processes,unfoldedProcesses)))
                    return true
                else continue
            }
        }
//No possible actions at currentNode
        return false
}
```

In function *buildCommunication* we will try to find a place in the graph for a node with the target network, that we have obtained after performing the communication action on the original network. The details of occurred communication are stored in the label and will be used as an edge in the graph between the current node and the new node.

First, *buildCommunication* checks if all processes in the current network have been already involved in any action, including the two processes that are involved in the current communication. If it is true, we mark the label as it was "flipped" and reset marking in the target network to the original state (meaning that only the terminated and livelocked processes will remain marked).

Then it checks if the node with the same network (as target network) and the same marking is already presented in the graph.

In the first case, if there is no such node, it attempts to add a new node to the graph. If it succeeds, it calls *buildGraph* with the new node as an argument. In case if *buildGraph* returns false, it removes from the graph the node that was just added. *buildCommunication* returns false in both cases, either it did not manage to add a new node to the graph, either the subsequent building of the graph wasn't successful.

In the second case, if the graph has already the node with network and marking equal to the target, then it needs only to add a new edge to this node without calling the *buildGraph* next, since the whole subgraph from that node was already built. And so there is no need to update any badNodes list. If *buildCommunication* manages to add the edge, it returns true, otherwise it fails.

```
1  fun buildCommunication
2    (targetNetwork: Network,label: InteractionLabel,currentNode: Node)
3    : Boolean {
4      val targetMarking = Marking(currentNode.marking)
5      targetMarking[label.sender] = true
6      targetMarking[label.receiver] = true
7
8  // if all processes reduced,reset the marking
9      if (targetMarking.values.all { it })
10         flipAndResetMarking(label,targetMarking,targetNetwork)
11
12 // check if a node with same network and marking is already in the graph
13     val node = findNodeInGraph(targetNetwork,targetMarking,currentNode)
14     if (node == null) {
15         val newNode = createNewNode(targetNetwork,label,currentNode,
               targetMarking)
16         if (addNodeAndEdgeToGraph(currentNode,newNode,label)) {
17             if (buildGraph(newNode))
18                 return true
19             else {
20                 removeNodeFromGraph(newNode)
21                 return false
```

```
            }
        } else
            return false
    } else
        return addEdgeToGraph(currentNode,node,label)
}
```

Function *buildConditional* resembles *buildCommunication* as it also tries to find the place in the graph for two networks that were obtained as a result of performing conditional action. It checks the marking of the processess in the target networks, and in case if they all are already marked, marks the labels as flipped and reset the networks' marking.

As in *buildCommunication*, it checks if there are already nodes in the graph with target networks and markings (for thenNetwork and elseNetwork). However, the logic of *buildConditional* is more complicated than of *buildCommunication*, since here it needs to operate with two nodes and there are more cases where something might go wrong.

If the node with thenNetwork and marking is not present, *buildConditional* adds it and proceed with building the graph. If *buildGraph* fails, new thenNode is removed from the graph and *buildConditional* is considered to be failed.

If thenNode is already present, *buildConditional* creates an edge to it from the current node. In case of success, it moves to the manipulation with the elseNode, otherwise fails.

We deal with the "else" case as well as with the one for "then". If there is no elseNode, it will be created and *buildGraph* will be called. If there is the node, the edge will be created. The difference here is that if for some reasons something fails (adding the node, *buildGraph*, or adding the edge) before failing itself, *buildConditional* removes the "then node" from the graph (see *garbageCollectThen*).

However, if dealing with both then and else branches went well, *buildConditional* does the relabeling of then end else nodes (see *relabel*) and exits with true.

```
fun buildConditional
  (thenNetwork: Network,thenLabel: ThenLabel,elseNetwork: Network,elseLabel
     : ElseLabel,currentNode: Node)
  : Boolean {
```

```
4       val targetMarking = Marking(currentNode.marking)
5       targetMarking[thenLabel.process] = true
6
7       if (targetMarking.values.all == true) {
8           flipAndResetMarking(thenLabel,targetMarking,thenNetwork)
9           flipAndResetMarking(elseLabel,targetMarking,elseNetwork)
10      }
11
12      //check if the then node with the same network and markings already
            exists in the graph
13      val thenNode = findNodeInGraph(thenNetwork,targetMarking,currentNode)
14      val newThenNode: Node
15
16      if (thenNode == null) {
17          newThenNode = createNewNode(thenNetwork,thenLabel,currentNode,
                targetMarking)
18          addNodeAndEdgeToGraph(currentNode,newThenNode,thenLabel)
19
20          if (buildGraph(newThenNode) == false) {
21              removeNodeFromGraph(newThenNode)
22              return false
23          }
24      } else {
25          newThenNode = thenNode
26          if (!addEdgeToGraph(currentNode,thenNode,thenLabel))
27              return false
28      }
29
30      //check if the else node with the same network and markings already
            exists in the graph
31      val elseNode = findNodeInGraph(elseNetwork,targetMarking,currentNode)
32      val newElseNode: Node
33      val garbageCollectThen = {
34          if (thenNode == null)
35              removeNodeFromGraph(newThenNode)
36          else
37              graph.removeEdge(currentNode,newThenNode)
```

```
38      }
39
40      if (elseNode == null) {
41          newElseNode = createNewNode(elseNetwork,elseLabel,currentNode,
                targetMarking)
42          addNodeAndEdgeToGraph(currentNode,newElseNode,elseLabel)
43
44          if (buildGraph(newElseNode) == false) {
45              garbageCollectThen()
46              removeNodeFromGraph(newElseNode)
47              return false
48          }
49      } else {
50          newElseNode = elseNode
51          if (!addEdgeToGraph(currentNode,newElseNode,elseLabel)) {
52              garbageCollectThen()
53              return false
54          }
55      }
56
57      relabel(newThenNode)
58      relabel(newElseNode)
59      return true
60  }
```

The *buildMulticom* function is in charge of finding a place in the graph for a network originated by execution of the multicom-action and it follows the similar flow of action as *builCommunication* and *buildConditiona*. It also starts with dealing with the marking and unfoldings, but since multi-action interaction involves more processes than single-action communication, it needs to do it for the processes from each action, information about which is stored in the labels. Than it folds back as usual the unfolded procedures that were not participating in communication and if all procedures were visited, it flips all markings. Next it checks if the node with the same network and markings already exists in the graph. If it does not, *buildMulticom* tries to create a new one, add it to the graph and runs the build graph again. Or if the node is already exist, it

tries to connect it and the current node with the edge. If any of this fails (building graph, adding node or adding edge), *buildMulticom* will fail too.

```
fun buildMulticom
  (actions: List<InteractionLabel>,currentNode: Node,processesCopy: Map<
     String,ProcessTerm>,unfoldedProcesses: Set<String>)
  : Boolean {
    val label = MulticomLabel(actions)
    val targetMarking = Marking(currentNode.marking)
    val targetNetwork = Network(processesCopy)
    for (label in label.labels) {
        targetMarking[label.sender] = true
        targetMarking[label.receiver] = true
        unfoldedProcesses.removeAll(arrayOf(label.sender,label.receiver))
    }
    fold(unfoldedProcesses,targetNetwork,currentNode)
    if (targetMarking.values.all == true)
        flipAndResetMarking(label,targetMarking,targetNetwork)
    val existingNode = findNodeInGraph(targetNetwork,targetMarking,
       currentNode)
    if (existingNode == null) {
        val newNode = createNewNode(targetNetwork,label,currentNode,
           targetMarking)
        addNodeAndEdgeToGraph(currentNode,newNode,label)
        return buildGraph(newNode)
    } else {
        if (addEdgeToGraph(currentNode,existingNode,label))
            return true
    }
    return false
}
```

As might be clear from the title, the *collectMulticomActions* function implements the second part of the algorithm for collecting multi-action interaction. As one of the arguments, it receives the list of waiting actions, and it will populate the list of the actions, while waiting list is not empty.

First, it takes out the first action from the waiting list (that contains at the beginning only one action). If the list of receivers contains the action's receiver, *collectMulticomActions* will throw an exception, since the multicom can not contain actions with the same receiver, and this means that the network is ill-formed. Then it adds the action to the list of actions, and the receiver of the action to the list of receivers. Next it finds the process term of a receiver process, checks that it is in the right format (see *fillWaiting* for details). If so, *fillWaiting* returns the process term (and also implicitly populates the list of waiting actions), then *collectMulticomActions* replaces the terms of sending and receiving with respect to the actions occurred.

```
1  fun collectMulticomActions
2    (waiting:List<InteractionLabel>,receivers: List<String>,actions: List<
       InteractionLabel>,processesCopy: Map<String,ProcessTerm>,currentNode:
       Node) {
3      while (waiting.isEmpty() == false) {
4          val label = waiting.first()
5          waiting.remove(label)
6          if (receivers.contains(label.receiver))
7              throw MulticomException()
8          actions.add(label)
9          receivers.add(label.receiver)
10         val receiver = label.receiver
11         val sender = label.sender
12         val receiverProcessTerm = processesCopy[receiver]
13         val marking = currentNode.marking.clone()
14         val newReceiverBehaviour = fillWaiting(waiting,actions,label,
               receiverProcessTerm.main,processesCopy[receiver].procedures,
               marking)
15         if (newReceiverBehaviour != null) {
16             val senderProcessTerm = processesCopy[sender]
17             val senderBehaviour = senderProcessTerm.main
18
19             processesCopy.replace(receiver,ProcessTerm(receiverProcessTerm.
                   procedures,newReceiverBehaviour))
20             val senderBehaviourContinuation = senderBehaviour.continuation
21             processesCopy.replace(sender,ProcessTerm(senderProcessTerm.
                   procedures,senderBehaviourContinuation))
22         }
```

```
23      }
24  }
```

3.5.4 Removing loops from the graph

Before building the choreography on our graph, we want to modify it to simplify the procedure. The graph might contain loops, because of the procedure invocations and our goal is to get rid of such loops.

We introduce the new type of a node, called *InvocationNode*. This node has the same data as the node with procedure invocation, however, it does not have an outgoing edge, and in this way will be not involved in any loop in the graph.

We call the node with procedure invocation, a recursive node. These nodes have more than one incoming edge (or exactly one, if we talk about the root node). We store such nodes in recursiveNodes map.

The first step of our graph unrolling algorithm is to collect all recursive nodes. Then for each of such nodes we do the following:

1. We create an invocation node and add it to the graph and to the list of invocation nodes
2. We collect all incoming edges of the recursive node. For each edge we find its source. We delete the edge and create the new one between the source and the invocation node (previously it was before the source and our recursive node).

In this way we prune all incoming edges to the recursive node and redirect them to the invocation nodes, making the structure of the graph acyclic.

We return as a result the list of invocation nodes, since we still need to know where procedure invocations took place to build the choreography.

```
1  fun unrollGraph
2    (root: Node)
3    : List<InvocationNode> {
4      val invocations = List<InvocationNode>()
5      var count = 1
```

```
    val recursiveNodes = Map<String,Node>()
    if (graph.incomingEdgesOf(root).size == 1)
        recursiveNodes[count++] = root
    for (node in graph.nodeSet()) {
        if (node is Node && graph.incomingEdgesOf(node).size > 1)
            recursiveNodes[count++] = node
    }
    for (pair in recursiveNodes) {
        val invocationNode = InvocationNode(pair.key,pair.value)
        graph.addnode(invocationNode)
        invocations.add(invocationNode)

        val incomingEdges = Set(graph.incomingEdgesOf(pair.value))
        for (edge in incomingEdges.) {
            val source = graph.getEdgeSource(edge)
            graph.removeEdge(edge)
            graph.addEdge(source,invocationNode,edge)
        }
    }
    return invocations
}
```

3.5.5 Building choreography

We start building the choreography in the *buildChoreography* function, which prepares the scene for the the *buildChoreographyBody*, where the main action happens.

The *buildChoreography* needs to know the root of the graph and the list of the invocation nodes (nodes with procedure invocation). Its first step would be checking if the root node is one of the invocation nodes, because in this case choreography body would contain only of one action - procedure invocation. If root node is not a procedure invocation itself, it runs *buildChoreographyBody* to collect all actions which will constitute the main procedure of a choreography.

Apart from the main procedure, it needs to build other choreography procedures, if any, so it executes *buildChoreographyBody* on the invocation nodes, which are known to be the procedures entry points.

Now when both main and common procedures are acquired, it can create a choreography out of them and return it as a result.

```
fun buildChoreography
  (root: Node,invocationNodes: List<InvocationNode>)
  : Choreography {
    val mainInvokers = invocationNodes.filter { it.node === root }
    val main =
        if (mainInvokers.isNotEmpty()) {
            ProcedureInvocation(mainInvokers.first().procedureName)
        } else {
            buildChoreographyBody(root)
        }

    val procedures = List<ProcedureDefinition>()
    for (procedureNode in invocationNodes) {
        procedures.add(ProcedureDefinition(procedureNode.procedureName,
           buildChoreographyBody(procedureNode.node),Set()))
    }
    return Choreography(main,procedures)
}
```

Function *buildChoreographyBody* is a recursive one. It takes a node in the graph and processes it depending on how many outgoing edges this node has.

If the node does not have any outgoing edges, we can expect two possible valid scenarios. Either it is a node with a network where all processes were terminated, and in this case *buildChoreographyBody* will create termination action in the choreography and return, as we have encountered the invocation node, so *buildChoreographyBody* will create the procedure invocation action and return. If none of this true, it means that the graph is not extractable.

If the node has one outgoing edge, it means there is an interaction. Since an interaction might be of three possible types, *buildChoreographyBody* needs to find out which of them is involved here

and to raise exception if none. The procedure is similar in case of communication or selection. Function *buildChoreographyBody* identifies the target node of the edge and recursively calls itself on it. Then it uses the information stored in the edge to create a communication or a selection action respectfully, using the value returned by calling itself on the action's continuation. In the case of multicom, it does the same for the target node, but here it processes the edge differently, since the edge of a multicom type contains several labels representing several asynchronous actions and it wants to collect all of them. Each of these labels might be of communication or selection type itself. As far as all multicom actions are collected, *buildChoreographyBody* can create the multicom with the result of calling itself on a target node as continuation.

If the node has two outgoing edges, it means that it is a conditional. Here, *buildChoreography-Body* calls itself on two nodes to which outgoing edges lead and which will represent then and else branches of a conditional it creates with them.

Since there is no possible action that has more than two possible ways of further development, *buildChoreographyBody* should raise an exception if there is a node with more than two outgoing edges.

```
1  fun buildChoreographyBody
2    (node: Node)
3    : ChoreographyBody {
4      val edges = graph.outgoingEdgesOf(node)
5      when (edges.size) {
6          0 -> {
7              when (node) {
8                  is Node -> {
9                      if (node.network.processes.all { it.value.main is
                           TerminationSP })
10                         return Termination()
11                     else
12                         throw IllegalStateException()
13                 }
14                 is InvocationNode ->
15                     return ProcedureInvocation(node.procedureName)
16                 else ->
17                     throw IllegalStateException()
```

```
18              }
19          }
20          1 -> {
21              val edge = edges.first()
22              when (edge) {
23                  is CommunicationLabel ->
24                      return CommunicationSelection(Communication(edge.sender
                            ,edge.receiver,edge.expression),
                            buildChoreographyBody(graph.getEdgeTarget(edge)))
25                  is SelectionLabel ->
26                      return CommunicationSelection(Selection(edge.receiver,
                            edge.sender,edge.label),buildChoreographyBody(graph.
                            getEdgeTarget(edge)))
27                  is MulticomLabel -> {
28                      val actions = List<Interaction>()
29                      for (label in edge.labels) {
30                          when (label) {
31                                  is SelectionLabel ->     actions.add(
                                        Selection(label.sender,label.receiver,
                                        label.label))
32                                  is CommunicationLabel -> actions.add(
                                        Communication(label.sender,label.
                                        receiver,label.expression))
33                          }
34                      }
35                      return Multicom(actions,buildChoreographyBody(graph.
                            getEdgeTarget(edge)))
36                  }
37                  is ConditionLabel ->
38                      throw IllegalStateException()
39              }
40          }
41          2 -> {
42              val thenLabel = edges.filterIsInstance<ThenLabel>().first()
43              val elseLabel = edges.filterIsInstance<ElseLabel>().first()
```

```
44              return Condition(thenLabel.process,thenLabel.expression,
                    buildChoreographyBody(graph.getEdgeTarget(thenLabel)),
                    buildChoreographyBody(graph.getEdgeTarget(elseLabel)))
45          }
46          else -> throw IllegalStateException()
47      }
48  }
```

3.5.6 Auxiliary methods

purgeNetwork

Function *purgeNetwork* is in charge of removing from the network the processes which in reality do no not participate in any action.

It takes each process in the network and checks if it was terminated or if the procedure it calls has its term terminated. If not, it adds the information about this process to the map of process names and terms. If there were no processes found satisfying this check, *purgeNetwork* takes a random process and adds it to the map. Finally it returns the map back.

```
1  fun purgeNetwork
2    (network: Network)
3    : Network {
4      val map = Map<ProcessName,ProcessTerm>()
5      network.processes.forEach { (processName, processTerm) ->
6          val exclude =
7              when(processTerm.main) {
8                  is TerminationSP -> true
9                  is ProcedureInvocationSP -> processTerm.procedures[
                       processTerm.main.procedure] is TerminationSP
10                 else -> false
11             }
12         if (!exclude)
13             map[processName] = processTerm
14     }
15     if (map.isEmpty()) {
16         val processName = network.processes.keys.first()
17         map[processName] = network.processes[processName]!!
18     }
19     return Network(map)
20 }
```

splitNetwork

There is a chance that the processes in the network might be organized in communicating groups which do not intersect with each other. In this case there is no point to treat them as a global entity, as this leads to storing more accompanying data and slowing down the extraction. So the goal of the *splitNetwork* network is to split the network to several not intersecting ones if possible. It works with the *getProcessSets* function under the hood, which is in charge of analyzing interactions between processes in the network and creating not intersecting sets of them.

First, *splitNetwork* receives the sets of intersecting processes with the the *getProcessSets* function. The *getProcessSets* creates the map between each process in the network and the set of processes it is interacting with which are being calculated in the *compute* function and then deduces the absence of intersections between these set. With the received sets, *splitNetwork* constructs the network for each of them, by finding in the original network the corresponding process term for each process in the set.

```
1  fun splitNetwork
2    (network: Network)
3    : Set<Network> {
4      val processSets = getProcessSets(network)
5      val networks = setOf<Network>()
6      for (processSet in processSets) {
7          val processes = mapOf<ProcessName,ProcessTerm>()
8          for (process in processSegetProcessSett) {
9              processes.put(process,network.processes[process])
10         }
11         networks.add(Network(processes))
12     }
13     return networks
14 }
15
16 fun getProcessSets
17   (network: Network)
18   : List<Set<String>> {
19     val map = Map<ProcessName,Set<ProcessName>>()
```

```
20      for (processName,processTerm in network.processes) {
21          map[processName] = compute(processTerm)
22      }
23      val processSets = List<Set<ProcessName>>()
24      val unprocessed = Set<ProcessName>(network.processes.keys)
25      while (unprocessed.isNotEmpty()) {
26          val component = Set<ProcessName>()
27          val processName = unprocessed.first()
28          component.add(processName)
29          unprocessed.remove(processName)
30          val visible = map[processName]
31          visible.add(processName)
32          while (visible != component) {
33              val intersection = (visible - component)
34              for (elem in intersection) {
35                  component.add(elem)
36                  unprocessed.remove(elem)
37                  visible.addAll(map[elem])
38              }
39          }
40          processSets.add(component)
41      }
42      return processSets
43  }
```

fold and unfold

We use the *unfold* function to replace the procedure invocation on the top of the process term with the procedure body. If the procedure body is a procedure invocation itself, the *unfold* will recursively repeat unfolding. If the unfolding was successful (and there was something to unfold), the *unfold* returns true, otherwise it fails. If there is no procedure term corresponding to the procedure name, an exception would be raised.

The *fold* function is dual to the *unfold* and is used to fold back the unfolded processes, means that for each process where procedure invocation on the top of the term was replaced by the procedure body, it puts back the procedure invocation.

```
fun unfold
  (p: String,processTerm: ProcessTerm)
  : Boolean {
    val processTermMain = processTerm.main
    if (processTermMain is ProcedureInvocationSP) {
        val processTermProcedure = processTermMain.procedure
        val procedureBehaviour = processTerm.procedures[
            processTermProcedure]
        processTerm.main = procedureBehaviour
//else throw Exception(Can't unfold the process)
        if (procedureBehaviour is ProcedureInvocationSP)
            unfold(p,processTerm)
        return true
    } else return false
}

fun fold
  (unfoldedProcesses: Set<String>,targetNetwork: Network,currentNode: Node)
      {
    for (process in unfoldedProcesses) {
        targetNetwork.processes[process].main = currentNode.network.
            processes.main
    }
}
```

findNodeInGraph

Function *findNodeInGraph* function helps us to find the node which is already presented in the graph and which shares the beginning of the choice path with the original node. First *findNodeInGraph* looks into the set of node hashes to see if there exists a node with the same hash (calculated on network and marking). Then it filters out all elements that are not predecessors of the node (node's choice path should start with the choice path of the nodes we are interested in, since that shows that they are predecessors of this node).

Finally, if there is any suitable nodes left, *findNodeInGraph* checks if any of them have the same network and marking and takes the first one (which is also the only one).

```
fun findNodeInGraph
  (network: Network,marking: Marking,node: Node)
  : Node {
    val existingNodes = nodeHashes[hash(network,marking)]
    filter existingNodes by { elem ->
        node.choicePath.startsWith(elem.choicePath)
    }
    return existingNodes.firstOrNull {
        it.network == network && it.marking == marking
    }
}
```

flipAndResetMarking

Function *flipAndResetMarking* is in charge of marking the edge between two nodes when all processes in the target node have been participated in at least one action already. It also resets the marking for such processes, leaving marked only terminated and livelocked processes.

```
fun flipAndResetMarking
  (label: ExtractionLabel,marking: Marking,network: Network) {
    label.flipped = true
    for (key in marking.keys) {
        marking[key] = network.processes[key].main is TerminationSP ||
            services.contains(key)
    }
}
```

doesNotContainInvocations and containsInvocation

Function *doesNotContainInvocations* checks if the process does not contain process invocation. It calls *containsInvocation*, which checks if the process term contains any invocation call. It is the recursive function which takes the first action in the term, and if it is not procedure invocation or termination, it calls itself on the continuation of the action (or continuations in case of several branches in label selection). In case it finds the procedure invocation, it returns true; if it reaches the end of the term, it returns false.

```
fun doesNotContainInvocations
  (pr: Behaviour)
  : Boolean {
    return !containsInvocation(pr)
}

fun containsInvocation
  (pr: Behaviour)
  : Boolean {
    when (pr) {
        is ProcedureInvocationSP -> return true
        is TerminationSP -> return false
        is SendSP -> return doesNotContainInvocations(pr.continuation)
        is ReceiveSP -> return doesNotContainInvocations(pr.continuation)
        is SelectionSP -> return doesNotContainInvocations(pr.continuation)
        is OfferingSP -> {
            for (label in pr.branches) {
                if (containsInvocation(label.value))
                    return true
            }
        }
        is ConditionSP -> {
            return containsInvocation(pr.elseBehaviour) ||
                containsInvocation(pr.thenBehaviour)
        }
    }
    return false
}
```

allTerminated

Function *allTerminated* checks if all processes in the network were terminated.

```
fun allTerminated
  (network: Map<String,ProcessTerm>)
  : Boolean {
    for (process in network) {
```

```
        if (process.value.main !is TerminationSP)
            return false
    }
    return true
}
```

findCommunication

Function *findCommunication* is needed to find and execute (if it exists) the interaction taking place between the process of interest and another process.

It takes the first action of a main procedure of a process term and checks its type. If it represents any kind of communication (sending/receiving, label offering/selecting), *findCommunication* takes the name of the process, with which the first one is interacting, and checks if it has the dual action on the top of its main procedure (for example, if the main procedure of the first process, e.g *q* has p!<e> on top, the p process has to have q? on top). If this condition was satisfied, *findCommunication* calls *consumeCommunication* (if there was sending/receiving interaction) or *consumeSelection* (if there was label selection/offering). *ConsumeCommunication* creates and returns a communication edge, which is a pair of the new version of the network with the communication being consumed and the label that stores information about the communication participants and passing message. *ConsumeSelection* function does the similar manipulation on network and label. Finally, the *findCommunication* returns the communication edge being produced by the *consumeCommunication* or *consumeSelection*, or returns *null*, if no interaction was found.

```
fun findCommunication
  (processes: ProcessMap,processName: String,processTerm: ProcessTerm)
  : CommunicationEdgeTo {
    val behaviour = processTerm.main
    when (behaviour) {
        is SendSP -> {
            val receiverTerm = processes[behaviour.process]
            if (receiverTerm.main is ReceiveSP && receiverTerm.main.sender
                == processName) {
```

```
                return consumeCommunication(processes,processTerm,
                    receiverTerm)
            }
        }
        is ReceiveSP -> {
            val senderTerm = processes[processTerm.main.process]
            if (senderTerm.main is SendSP && senderTerm.main.receiver ==
                processName) {
                return consumeCommunication(processes,senderTerm,
                    processTerm)
            }
        }
        is SelectionSP -> {
            val offerTerm = processes[behaviour.process]
            if (offerTerm.main is OfferingSP && offerTerm.main.sender ==
                processName) {
                return consumeSelection(processes,offerTerm,processTerm)
            }
        }
        is OfferingSP -> {
            val selectTerm = processes[behaviour.process]
            if (selectTerm.main is SelectionSP && selectTerm.main.receiver
                == processName) {
                return consumeSelection(processes,processTerm,selectTerm)
            }
        }
    }

    return null
}

typealias CommunicationEdgeTo = Pair<Network, InteractionLabel>

fun consumeCommunication
  (processes: ProcessMap,senderTerm: ProcessTerm,receiverTerm: ProcessTerm)
  : CommunicationEdgeTo {
    val processCopy = copyProcesses(processes)
```

```
41      val receiverName = senderTerm.main.receiver
42      val senderName = receiverTerm.main.sender
43
44      processCopy.replace(senderName,ProcessTerm(senderTerm.procedures,
            senderTerm.main.continuation))
45      processCopy.replace(receiverName,ProcessTerm(receiverTerm.procedures,
            receiverTerm.main.continuation))
46
47      val label = CommunicationLabel(senderName,receiverName,senderTerm.main.
            expression)
48      return CommunicationEdgeTo(Network(processCopy),label)
49  }
50
51  fun consumeSelection
52    (processes: ProcessMap,offerTerm: ProcessTerm,selectTerm: ProcessTerm)
53    : CommunicationEdgeTo {
54      val processCopy = copyProcesses(processes)
55      val selectionProcess = offerTerm.main.process
56      val offeringProcess = selectTerm.main.process
57      val offeringBehavior = offerTerm.main.branches[selectTerm.main.label]
58  //throw exception if selectionProcess is trying to select label, which does
        not offer it
59      processCopy.replace(offeringProcess,ProcessTerm(offerTerm.procedures,
            offeringBehavior))
60      processCopy.replace(selectionProcess,ProcessTerm(selectTerm.procedures,
            selectTerm.main.continuation))
61      val label = SelectionLabel(offeringProcess,selectionProcess,selectTerm.
            main.label)
62
63      return CommunicationEdgeTo(Network(processCopy),label)
64  }
```

findConditional

Function *findConditional* is similar to *findCommunication*. It checks if the main procedure of the process has a conditional on top. In case of success, it creates and returns conditional edges

which is the object consisted of two new networks for then and else branches after the conditional being consumed, and two labels — *then* end *else* label — with the information of a process name and conditional expression.

```
data class ConditionEdgesTo(
    val thenNetwork: Network,
    val thenLabel: ThenLabel,
    val elseNetwork: Network,
    val elseLabel: ElseLabel)

class ThenLabel(val process: String, val expression: String):
    ConditionLabel()
class ElseLabel(val process: String, val expression: String):
    ConditionLabel()

fun findConditional(processes: ProcessMap,processName: String,processTerm:
    ProcessTerm): ConditionEdgesTo? {
        if (processTerm.main !is ConditionSP)
            return null

        val conditional = processTerm.main as ConditionSP
        val thenProcessesMap = Map<String,ProcessTerm>(processes)
        val elseProcessesMap = Map<String,ProcessTerm>(processes)

        thenProcessesMap.replace(processName,ProcessTerm(processTerm.
            procedures,conditional.thenBehaviour))
        elseProcessesMap.replace(processName,ProcessTerm(processTerm.
            procedures,conditional.elseBehaviour))

        return ConditionEdgesTo(Network(thenProcessesMap),ThenLabel(
            processName,conditional.expression),Network(elseProcessesMap),
            ElseLabel(processName,conditional.expression))
    }
```

createNewNode

Function *createNewNode* is in charge of preparing a new node, before it can be added to the graph. As it was mentioned earlier, apart from storing a network, each node also stores auxiliary information about node marking, choice path and the set of previously encountered nodes (the network and the marking are already calculated at this point and provided in parameters).

To calculate the choice path of the node, we need to know if the node is the result of a conditional or an interaction, which can be determined by the type of its label: if it is of type *thenLabel* or *elseLabel*, it is a conditional otherwise it is an interaction. Nodes resulting from the conditional get an extra "0" or "1" to their choice path for then and else nodes respectively or remains equal to the current node choice path, in case of interaction.

To calculate the set of previously encountered nodes ("bad nodes"), we need to know if all processes in the network were visited (or terminated or livelocked) by this time. To do this we check the edge between the node and the node's predecessor. If its label was "flipped", it means that all processes in the node's network have been already visited and we do not need to remember the node's predcessors. Otherwise the set of bad nodes will contain the predecessor and all nodes that it has in the set of bad nodes itself.

Finally, *createNewNode* creates the node with the network, marking and calculated bad nodes and choice path.

```
1  fun createNewNode
2    (targetNetwork: Network,label: ExtractionLabel,predecessorNode: Node,
         marking: Map<ProcessName,Boolean>)
3    : Node {
4      val str = when (label) {
5          is ThenLabel -> predecessorNode.choicePath + 0
6          is ElseLabel -> predecessorNode.choicePath + 1
7          else -> predecessorNode.choicePath
8      }
9      val badNodes = Set<Int>()
10     if (!label.flipped) {
11         badNodes.addAll(predecessorNode.badNodes)
12         badNodes.add(predecessorNode.id)
```

```
    }
    return ConcreteNode(targetNetwork,choicePath,nextNodeId(),badNodes,
        marking)
}
```

addToMap and removeFromMap

Function *addToMap* is in charge of calculating the hash of a node and adding it to the global map of the node hashes. Since there might be several nodes with the same hashes, *nodeHashes* uses a hash as a key to the list of nodes with this hash.

The hash is based on a both network and network's marking. It is important to consider marking, look for example at the following network.

```
  a1 {def X {b1!<e>; X} main {X}}
| b1 {def X {a1?; X} main {X}}
| a2 {def X {b2!<e>; X} main {X}}
| b2 {def X {a2?; X} main {X}}
```

After the execution of the first action, no matter between *a1* and *b1* processes or between *a2* and *b2*, the network will look the same and have the same hash if calculated only based on a network itself, so it is important to hash also some evidence of processes being executed.

After the calculating the hash, *addToMap* checks if the *nodeHashes* already it, and in this case adds the node to the list of nodes with corresponding hash. If the hash was not found, *addToMap* creates a new record, with hash as a key and the new list with the node as value.

Its pair, the *removeFromMap* function is in charge of removing the node from the map of nodes hashes.

```
fun addToMap
  (newNode: Node) {
    val hash = hash(newNode.network,newNode.marking)
    nodeHashes.compute(hash) { value ->
        if (value == null) {
            val array = List<Node>()
            array.add(newNode)
```

```
        } else
            value.add(newNode)
    }
}

fun removeFromMap
  (newNode: Node) {
    nodeHashes.remove(hash(newNode.network,newNode.marking))
}
```

addToChoicePathMap and removeFromChoicePathMap

Functions *addToChoicePathMap* and *removeFromChoicePathMap* are the same as *addToMap* and *removeFromMap* with only difference that here they deal with the map of choice pathes instead of hashes.

```
fun addToChoicePathMap
  (node: Node) {
    choicePaths.compute(node.choicePath) { value ->
        if (value == null) {
            val array = List<Node>()
            array.add(node)
        } else
            value.add(node)
    }
}

fun removeFromChoicePathMap
  (node: Node) {
    choicePaths[node.choicePath].remove(node)
}
```

addNodeAndEdgeToGraph and removeNodeFromGraph

Function *addNodeAndEdgeToGraph* processes adding a new node and an edge between the predcessor's node and the new node to the graph.

If it manages to add both the node and the edge, it logs information about the new node to the maps of nodes choices and hashes and exits with true. Otherwise it fails, with only difference that if it fails on adding the edge, it also removes the recently added node.

AddNodeAndEdgeToGraph has a twin, which is *removeNodeFromGraph*. It removes the node from the graph and from the maps of the nodes choice paths and hashes. The edge is being removed implicitly.

```
fun addNodeAndEdgeToGraph
  (currentNode: Node,newNode: Node,label: ExtractionLabel)
  : Boolean {
    if (graph.addnode(newNode)) {
        if (graph.addEdge(currentNode,newNode,label)) {
            addToChoicePathMap(newNode)
            addToMap(newNode)
            return true
        } else graph.removenode(newNode)
    }
    return false
}

fun removeNodeFromGraph
  (removingNode: Node) {
    graph.removenode(removingNode)
    removeFromMap(removingNode)
    removeFromChoicePathMap(removingNode)
}
```

checkLoop and addEdgeToGraph

Function *checkLoop* is one of the most important in the extraction algorithm, because it checks that building the graph goes right. In particular, it looks if all actions in a recursive procedure were executed before closing the loop. If do not do that, it may lead to graph anomalies and no correct choreography could be extracted from it.

We check for presence of a bad loop when we need to add an edge to an already existent node, which is being done in *addEdgeToGraph*. Its logic is quite simple: it calls *checkLoop* and if it returns true (no bad loop present), it returns the result of adding the node to the graph. Otherwise, if *checkLoop* returns false, *addEdgeToGraph* increases the bad loop counter and returns false.

In turn *checkLoop* first checks if the label of the edge between source and target nodes was flipped, meaning that all processes in the network have been visited (or terminated or livelocked). If it is not the case, it checks if the target node is the same as the source node, meaning that the loop leads to itself, and the node's network contains processes that were not executed, so *checkLoop* fails. Otherwise it checks if the target node is in the set of the nodes previously encountered by the source node. In this case *checkLoop* also fails, since it is clear that not all processes were visited at this point, otherwise it returns true.

```
fun addEdgeToGraph
  (sourceNode: Node,targetNode: Node,label: ExtractionLabel)
  : Boolean {
    if (checkLoop(sourceNode,targetNode,label)
        return graph.addEdge(sourceNode,targetNode,label)
    badLoopCounter++
    return false
}

fun checkLoop
  (sourceNode: Node,targetNode: Node,label: ExtractionLabel)
  : Boolean {
    if (label.flipped) return true
    if (targetNode === sourceNode) return false
    return (!sourceNode.badNodes.contains(targetNode.id))
}
```

relabel

If we mange to build graph for both of conditional branches, we do not need to be worried about having different choice paths for each of two branches, so we use the *relabel* function to remove the last digit of their choice paths differentiating it one from another.

```
fun relabel
  (node: Node) {
    val key = node.choicePath.dropLast(1)
    addToChoicePathMap(Node(node.network,key,node.id,node.badNodes,node.
        marking))
    removeFromChoicePathMap(node)
}
```

copyAndSortProcesses

Function *copyAndSortProcesses* returns the copy of the node's network with all processes sorted with respect to the extraction strategy. There are several possible extraction strategies presented and each of them has its own logic for process sorting implemented in the *copyAndSort* function.

```
fun copyAndSortProcesses
  (node: Node)
  : Map<String,ProcessTerm> {
    return strategy.copyAndSort(node)
}

enum class ExtractionStrategy {
    InteractionsFirst { override fun copyAndSort( ... ) { ... } },
    ConditionsFirst { ... },
    UnmarkedFirst { ... },
    LongestFirst { ... },
    Random { ... },
    ...
    Default { ... };
    abstract fun copyAndSort(node: Node): Map<String, ProcessTerm>
}
```

fillWaiting

Function *fillWaiting* is part of the algorithm for creating a multicom. It checks if the continuation of the receiving process to be sure that it is in the form $a_1;...;a_k;r?;B$ where each a_i is either the sending of a value or a label selection. This is the recursive function which processes the continuation action by action. If the action is of interaction type (sending or selection), *fillWaiting* calls itself on its continuation until it reaches the receiving or offering action. In this case, it checks it this receiving/offering is the one we are interested in (with the same sender and of the same type). If this is true, it means the continuation is well-formed and *fillWaiting* returns continuation of the receiver. Then it creates a new label, dual to the original one and put it to the list of waiting actions (if it is not in the list of actions already), then it returns the modified receiver term, removing the found receiving/offering action from it. If during the processing *fillWaiting* meets the procedure invocation, it unfolds it (not forgetting to track on markings) and calls itself on the unfolded term. If the receiver's term violates the required form, *fillWaiting* will return *null*.

```
1  fun fillWaiting
2    (waiting: List<InteractionLabel>,actions: List<InteractionLabel>,label:
         InteractionLabel,receiverProcessTermMain: Behaviour,receiverProcesses:
          Map<String,Behaviour>,marking: Map<ProcessName,Boolean>)
3    : Behaviour {
4      when (receiverProcessTermMain) {
5          is SendSP -> {
6              val sendingContinuation = fillWaiting(waiting,actions,label,
                   receiverProcessTermMain.continuation,receiverProcesses,
                   marking)
7              if (sendingContinuation != null) {
8                  val newLabel = CommunicationLabel(label.receiver,
                       receiverProcessTermMain.receiver,receiverProcessTermMain
                       .expression)
9                  if (!actions.contains(newLabel)) waiting.add(newLabel)
10                 return SendSP(sendingContinuation,newLabel.receiver,
                       newLabel.expression)
11             } else {
12                 return null
13             }
```

```
14          }
15          is SelectionSP -> {
16              val selectionContinuation = fillWaiting(waiting,actions,label,
                    receiverProcessTermMain.continuation,receiverProcesses,
                    marking)
17              if (selectionContinuation != null) {
18                  val newLabel = SelectionLabel(label.receiver,
                        receiverProcessTermMain.receiver,receiverProcessTermMain
                        .label)
19                  if (!actions.contains(newLabel)) waiting.add(newLabel)
20                  return SelectionSP(selectionContinuation,label.receiver,
                        label.expression)
21              }
22          }
23          is ReceiveSP -> if (label.sender == receiverProcessTermMain.sender
                && label is CommunicationLabel) return receiverProcessTermMain.
                continuation
24          is OfferingSP -> if (label.sender == receiverProcessTermMain.sender
                && label is SelectionLabel) return receiverProcessTermMain.
                branches[label.label]
25          is ProcedureInvocationSP -> {
26              val newProcessTerm = ProcessTerm(receiverProcesses,
                    receiverProcessTermMain)
27              unfold(label.receiver,newProcessTerm)
28              marking[label.receiver] = true
29              return fillWaiting(waiting,actions,label,newProcessTerm.main,
                    receiverProcesses,marking)
30          }
31      }
32      return null
33  }
```

createInteractionLabel

Function *createInteractionLabel* is used in the algorithm for finding a multicom. It checks if a process has sending or selection action on top of its main procedure and creates the communication-

or selection label correspondingly with the information about the processes involved into action and expression/label. If the action is not of an interaction type, *createInteractionLabel* returns *null*.

```
fun createInteractionLabel
  (processName: String,processes: Map<String,ProcessTerm>)
  : InteractionLabel {
    val processTerm = processes[processName].main
    when (processTerm) {
        is SendSP -> {
            return CommunicationLabel(processName,processTerm.receiver,
                processTerm.expression)
        }
        is SelectionSP -> {
            return SelectionLabel(processName,processTerm.receiver,
                processTerm.label)
        }
        else -> null
    }
}
```

Miscellaneous

There are also plenty of rather standard functions to manipulate with nodes and networks: copy, equals, hash code calculating, creating the new node id, etc.

```
1  fun copy(): Node {
2      Node(network.copy(),choicePath,id,Set(badNodes),marking.clone() as Map<
           ProcessName,Boolean>)
3  }
4
5  fun equals(other: Node): Boolean {
6      id == other.id
7  }
8
9  fun hashCode(): Int {
10     network.hashCode() + 31 * choicePath.hashCode() + 31 * 31 * id + 31 * 3
           1 * 31 * badNodes.hashCode() + 31 * 31 * 31 * 31 * marking.hashCode
           ()
11 }
12
13 fun nextNodeId(): Int {
14     return nodeIdCounter++
15 }
16
17 fun hash(network: Network,marking: Marking): Int {
18     network.hashCode() + 31 * marking.hashCode()
19 }
20
21 fun copyProcesses(processes: Map<String,ProcessTerm>): Map<String,
       ProcessTerm> {
22     val copy = Map<String,ProcessTerm>()
23     processes.forEach { t,u -> copy[t] = u.copy() }
24     return copy
25 }
```

3.6 Collecting statistics

3.6.1 Process

One of the purposes of our research was analyzing how extraction procedure depends on the extraction strategies and the structure of the network.

The process flow of the statistic collection is the following:

1. Generating files with choreographies (we start from the choreographies because we want to see how a structure of a choreography impacts the extraction).
2. Projecting choreographies into networks and collecting projection statistics
3. Extracting choreographies and collecting extraction statistics

Generating files with choreographies

Extraction procedure depends on a structure of the choreography, so we have generated choreographies with different structure varying the number of communications (50-2100, with the step of 50), processes (5-100, with the step of 5), conditionals (0-40, with the step of 10), and procedures (0-15, with the step of 5).

Data on the networks

As projection statistics we were interested in the following parameters: number of actions, processes, procedures, and conditionals; the minimum, maximum and average length of processes, procedures, conditionals, and the number of processes with conditionals.

Extraction statistics

As extraction statistics we collect the extraction time, amount of nodes and bad loops in the SEG, the length of the main procedure, the number of procedures, the minimum, maximum and average length of procedures. We focused mainly on procedural statistics because procedures

make choreographies and SEG interesting for studying. Plain choreographies without recursion show linear statistical data; on the contrary, recursive choreographies extracted with different strategies make statistics vary in time, the number of nodes in the graph, and the number of bad loops encountered.

Extraction statistics returns as the result of *extract* function (we have omitted discussing it earlier for the separation of concerns).

3.6.2 Results

Considering the amount of parameters, we have obtained a massive number of results. Not all of them were meaningful and having an original point. We have collected the noteworthy pieces and divided them into four groups by the structure of original choreography. Each plot shows several functions with one of the choreography parameters for X-axis and extraction statistics parameter on the Y-axis. Functions differ as we ran the extraction with different strategies described in section 3.5.1.

For the tests with different number of processes and actions, we provide only timing analysis. They do not contain any procedures, so no bad loops were created, and they do not have conditionals, so the impact on the graph size is linear.

Communication

These plots are generated for the choreographies with the varying parameter of communications and showing its impact on the time. fig. 3.12 presents only four strategies from our set, since not all of them are making sense for these type of tests (e.g, we have omitted all unmarked strategies, as well as ConditionalFirst, because there are no conditional actions and procedures in these choreographies).

fig. 3.12a shows the behavior of all tested strategies. Extraction made with `ShortestFirst` and `LongestFirst` strategies grows quadratically in time, and `Random` and `InteractionFirst` approximately shows the complexity between quadratic and linear. However, closer look on the fig. 3.12b and fig. 3.12c shows the quadratic complexity for both cases. Better behav-

ior of Random and InteractionFirst strategies on the first figure is the result of avoiding the time overheads on calculating the length of the process intrinsic to ShortestFirst and LongestFirst strategies.

Number of processes

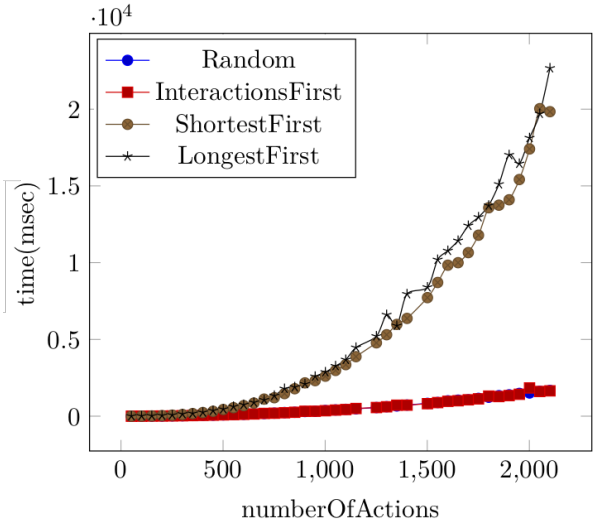
For the next tests, we varied the number of processes in a network. We have used the same strategies as in the tests with the varying amount of actions following the same reasoning (these tests do not contain procedures or conditionals).

fig. 3.13 shows the linear complexity of each extraction strategy. Here we see a similar pattern for LongestFirst and ShortestFirst strategies behaving worse than Random and InteractionFirst. However, LongestFirst and ShortestFirst fig. 3.13c, has the worst timing in the extraction with lesser numbers of processes, which does not hold for the Random and InteractionFirst strategies in fig. 3.13b. We explain it by the nature of the test: an increasing number of processes requires more time of processing, making the time spent on calculating the length less significant.

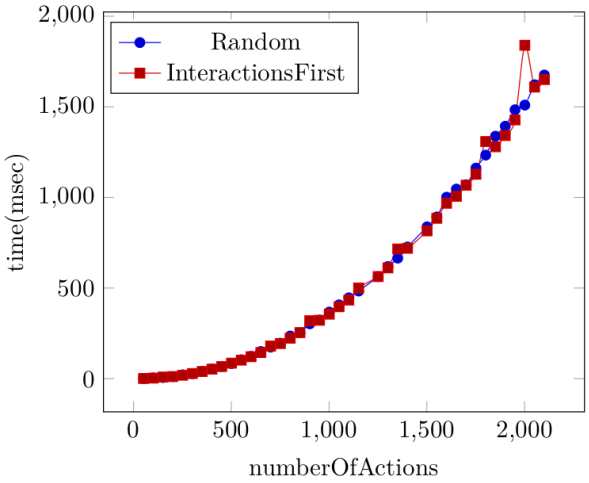
Number of conditionals

In the following tests, we were varying the number of conditionals. Here, except for already mentioned strategies, we add the ConditionsFirst.

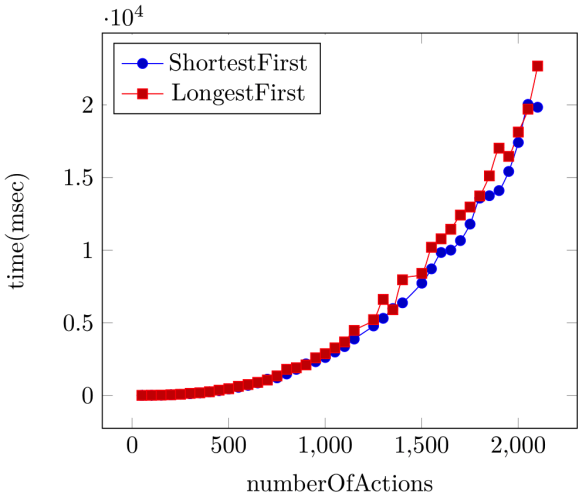
Figure 3.14a shows the exponential complexity of ConditionsFirst and LongestFirst strategies. Following ConditionsFirst strategy leads to doubling the size of the graph each time the conditional is encountered, which takes more time to process. Currently, we do not know the causes of bad complexity results of the LongestFirst strategy, which opens possibilities for future work. Figure 3.14b shows the linear complexity and tells us that the test's time complexity is not related to the amount of nodes. Random and InteractionFirst strategies show the complexity between quadratic and exponential, but we believe we do not have enough points to evaluate it precisely enough.



(a) *All extraction strategies*

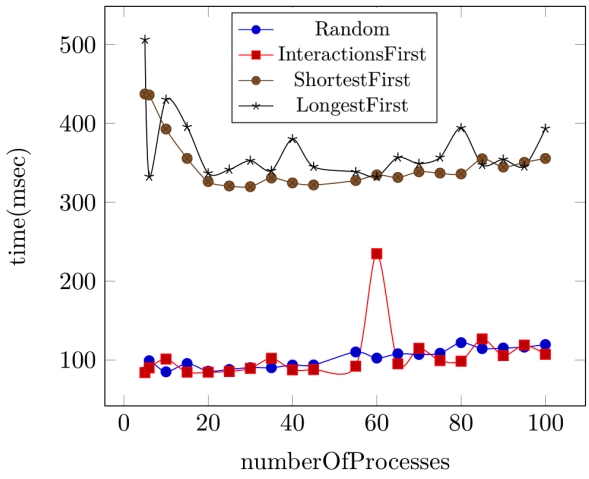


(b) *Random and IntercctionFirst strategies*

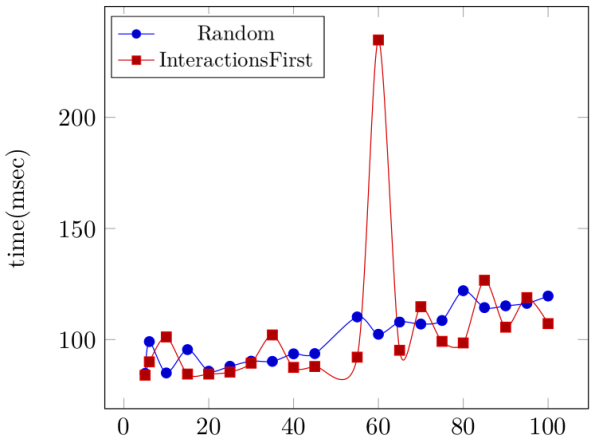


(c) *ShortestFirst and LongestFirst strategies*

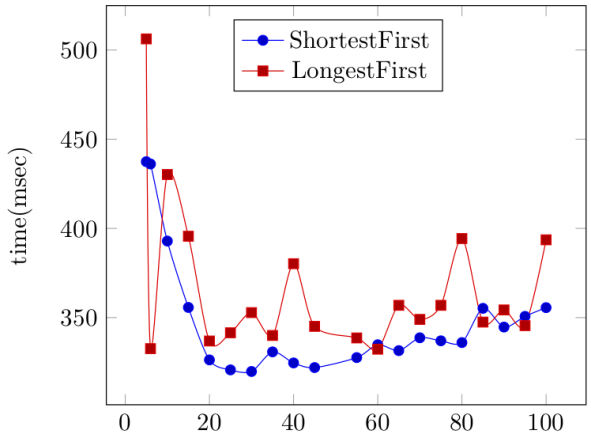
Figure 3.12: *Number of actions to time*



(a) *All extraction strategies*

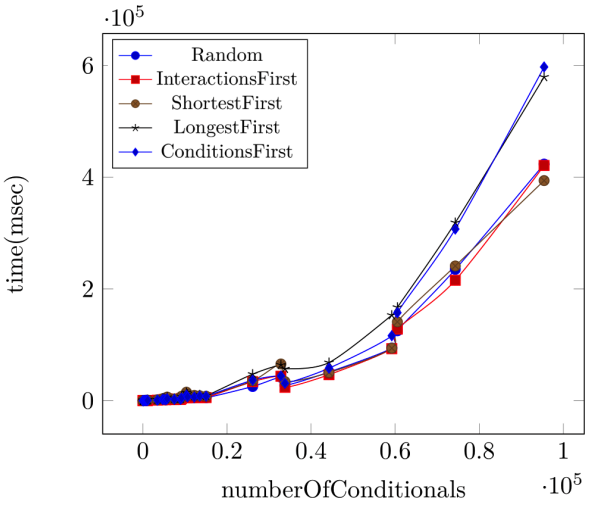


(b) *Random and IntercctionFirst strategies*

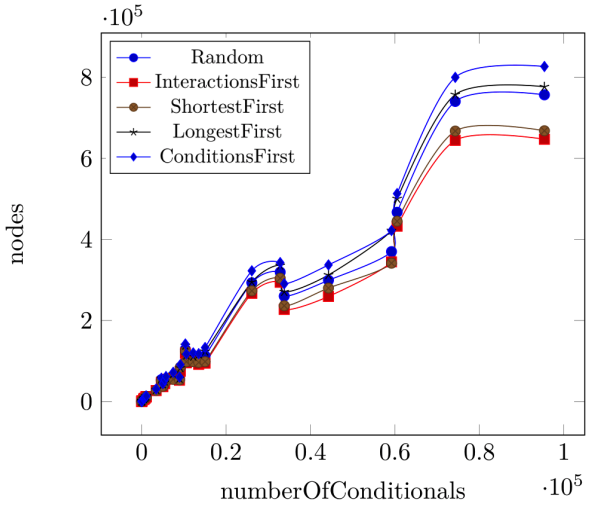


(c) *ShortestFirst and LongestFirst strategies*

Figure 3.13: *Number of processes to time*



(a) *Time statistics*



(b) *Nodes statistics*

Figure 3.14: *Number of conditionals to time and the number of nodes*

Conditionals and procedures

In our last set of tests, we are varying both conditionals and procedures. Here we add strategies prioritizing unmarked processes first: `UnmarkedFirst`, `UnmarkedThenCondition`, and `UnmarkedThenRandom`. The results we have obtained do not always make sense and leave us with a lot of open questions.

Figure 3.15 displays the result of increasing the number of conditionals. They show the worse result of `LongestFirst` and `ShortestFirst` strategies comparing to others, but we can not explicitly identify the cause of this behavior, since the test include two varying parameters, and one might skew the result of another. Figure 3.15b shows that it does not relate to the amount of nodes, as in the case in fig. 3.14b.

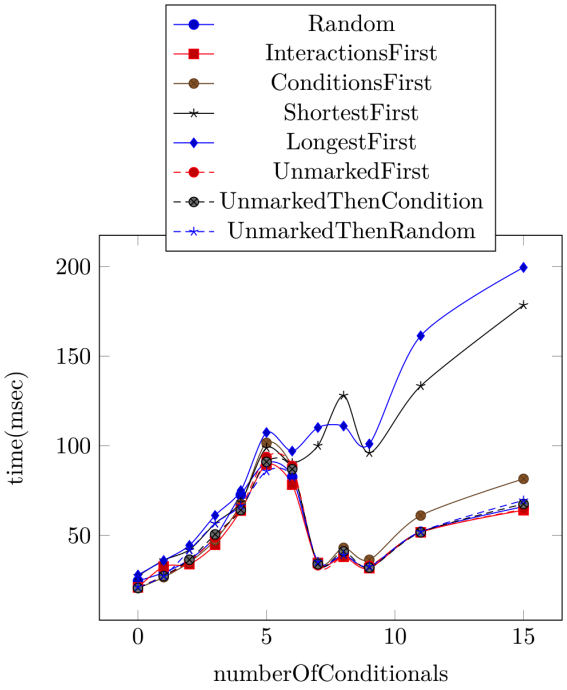
Figure 3.16a shows the statistics with an increasing number of procedures. Here, `LongestFirst` and `ShortestFirst` strategies behave worse than others with a lower number of procedures. fig. 3.16b shows a peculiar pattern holding in all strategies: the amount of nodes rises linearly and then goes down. We explain these results as follows: since the amount of procedures is rising, but the size of the network remains the same, the procedure body becomes smaller, and during the extraction of such networks, we generate less nodes.

Both fig. 3.15 and fig. 3.16 show that the strategies that prioritize the unmarked processes are not superior to the others.

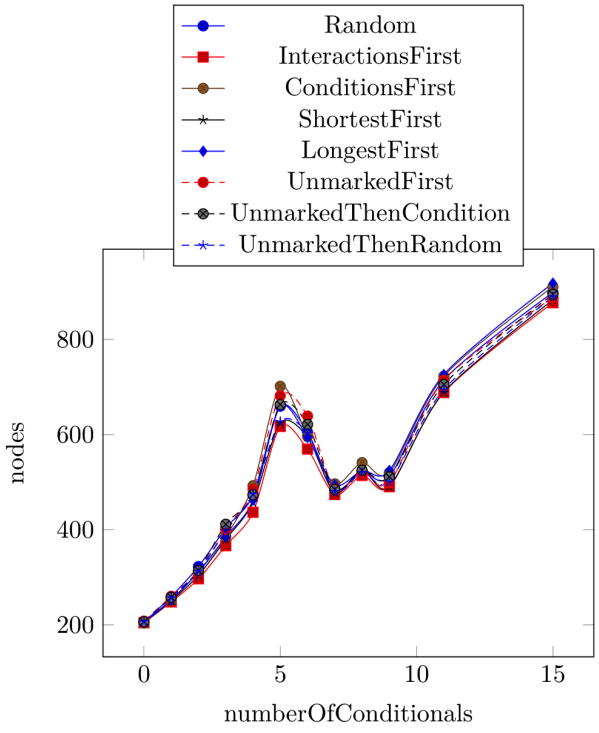
3.7 Future work

3.7.1 Refine strategies

In all tests, `ShortestFirst` and `LongestFirst` strategies show significant from other strategies growth in time. Depending on the nature of the test, this growth happens in different moment (e.g., for varying number of actions the time grows as the amount of actions grow, and for varying number of processes the time is higher with lesser amount of processes). But the reason of this behavior remains the same: the overhead of calculating the size of the process. To eliminate this

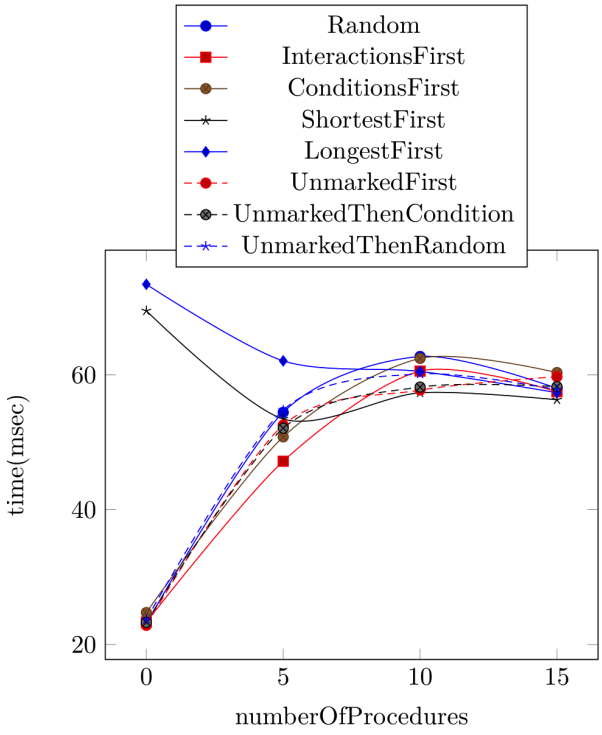


(a) *Time statistics*

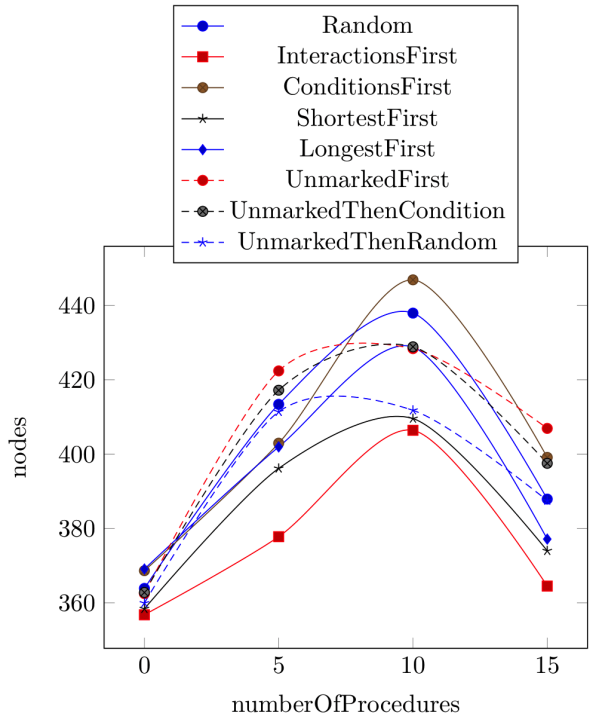


(b) *Nodes statistics*

Figure 3.15: *Number of conditionals to time and to number of nodes*



(a) *Time statistics*



(b) *Nodes statistics*

Figure 3.16: *Number of procedures to time and to number of nodes*

effect, we would like to augment the network with this information before the extraction so that we can save time in the future.

The fact that some of the tests have produced the results we could not explain means that our choice of strategies and parameters was not always optimized and grained enough. We believe that working in this direction could give us more exciting results with the potential to improve the efficiency of the extraction. For example, we could grain the set of tests with varying amounts of conditionals and procedures, but similar structure in general, and look at how their behavior changes as we change their ratio.

3.7.2 Static analysis on networks

Since some tests have shown that the structure of the network can impact on choice of the best extraction strategy, we would like to enrich our implementation with simple static analysis. For example, if a network contains a comparatively high number of conditionals in processes, we could avoid using the `ConditionalFirst` and `LongestFirst` strategies as they showed the worst results.

3.7.3 Treating unextractable networks

Our current implementation does not count the fact that some of the networks might be unextractable and do not terminate in the time allocated for the extraction process. Treating them similarly as the extractable ones can significantly skew the time in the tests (as it happens for test with varying number of procedures and conditionals). We need to change the extraction implementation in such manner, that we could understand the reason of the time grow, was it, for example, because of the backtracking in case of encountering a bad loop, or because the test has failed.

3.8 Conclusions

In this chapter, we have presented the implementation of the extraction algorithm defined by Cruz-Filipe et all. We have discussed optimizations to the algorithm, discovered during the implementation process. We have provided the empirical analysis of the implementation based on the thorough set of tests with different extraction strategies. We have confirmed that in the worst-case efficiency of the algorithm corresponds to the one claimed in [73]. Finally, we have proposed the directions of future work, including the fine-grained treatment of extraction strategies and implementation of network static analysis.

Chapter 4

Ephemeral Data Handling in Microservices

Abstract

In modern application areas for software systems — like eHealth, the Internet-of-Things, and Edge Computing — data is encoded in heterogeneous, tree-shaped data-formats, it must be processed in real-time, and it must be ephemeral, i.e., not persist in the system. While it is preferable to use a query language to express complex data-handling logics, their typical execution engine, a database external from the main application, is unfit in scenarios of ephemeral data-handling. A better option is represented by integrated query frameworks, which benefit from existing development support tools (e.g., syntax and type checkers) and execute within application memory. In this paper, we propose one such framework that targets document-oriented queries for data shaped in tree format. We formalize an instantiation of MQuery, a sound variant of the widely-used MongoDB query language, which we implemented in the Jolie language. Jolie programs are microservices, the building blocks of modern software systems. Moreover, since Jolie supports native tree data-structures and automatic management of heterogeneous data-encodings, we can provide a uniform way to use MQuery on any data-format supported by the language.

4.1 Introduction

Modern application areas for software systems—like eHealth [157], the Internet of Things [30], and Edge Computing [174]—need to address two requirements: velocity and variety [135]. Velocity concerns managing high throughput and real-time processing of data. Variety means that data might be represented in heterogeneous formats, complicating their aggregation, query, and storage. Recently, in addition to velocity and variety, it has become increasingly important to consider ephemeral data handling [179, 173], where data must be processed in real-time but not persist — ephemeral data handling can be seen as the opposite of dark data [180], which is data stored but not used. The rise of ephemeral data is due to scenarios with heavy resource constraints (e.g., storage, battery) — as in the Internet of Things and Edge Computing — or new regulations that may limit what data can be persisted, like the GDPR [148] — as in eHealth.

Programming data handling correctly can be time consuming and error-prone with a general-purpose language. Thus, often developers use a query language, paired with an engine to execute them [63]. When choosing the query execution engine, developers can either A) use a database management system (DBMS) executed outside of the application, or B) include a library that executes queries using the application memory.

Approach A) is the most common. Since the early days of the Web, programmers integrated application languages with relational (SQL-based) DBMSs for data persistence and manipulation [190]. This pattern continues nowadays, where relational databases share the scene with new NoSQL [135] DBMSs, like MongoDB [142] and Apache CouchDB [27], which are document-oriented. Document-oriented databases natively support tree-like nested data structures (typically in the JSON format). Since data in modern applications is typically structured as trees (e.g., JSON, XML), this removes the need for error-prone encoding/decoding procedures with table-based structures, as in relational databases. However, when considering ephemeral data handling, the issues of approach A) overcome its benefits even if we consider NoSQL DBMSs:

① Drivers and Maintenance. An external DBMS is an additional standalone component that needs to be installed, deployed, and maintained. To interact with the DBMS, the developer needs to import in the application specific drivers (libraries, RESTful outlets). As with

any software dependency, this exposes the applications to issues of version incompatibility [120].

② Security Issues. The companion DBMS is subject to weak security configurations [46] and query injections, increasing the attack surface of the application.

③ Lack of Tool Support. Queries to the external DBMS are typically black-box entities (e.g., encoded as plain strings), making them opaque to analysis tools available for the application language (e.g., type checkers) [63].

④ Decreased Velocity and Unnecessary Persistence. Integration bottlenecks and overheads degrade the velocity of the system. Bottlenecks derive from resource constraints and slow application-DB interactions; e.g., typical database connection pools [183] represent a potential bottleneck in the context of high data-throughput. Also, data must be inserted in the database and eventually deleted to ensure ephemeral data handling. Overheads also come in the form of data format conversions (see item ⑤).

⑤ Burden of Variety. The DBMS typically requires a specific data format for communication, forcing the programmer to develop ad-hoc data transformations to encode/decode data in transit (to insert incoming data and returning/forwarding the result of queries). Implementing these procedures is cumbersome and error-prone.

On the other side, approach B) (query engines running within the application) is less well explored, mainly because of the historical bond between query languages and persistent data storage. However, it holds potential for ephemeral data handling. Approach B) avoids issues ① and ② by design. Issue ③ is sensibly reduced, since both queries and data can be made part of the application language. Issue ④ is also tackled by design. There are less resource-dependent bottlenecks and no overhead due to data insertions (there is no DB to populate) or deletions (the data disappears from the system when the process handling it terminates). Data transformation between different formats (item ⑤) is still an issue here since, due to variety, the developer must convert incoming/outgoing data into/from the data format supported by the query engine. Examples of implementations of approach B) are LINQ [136, 63] and CQEngine [152]. While LINQ and CQEngine grant good performance (velocity), variety is still an issue. Those proposals either assume an SQL-like query language or rely on a table-like format, which entail continuous,

error-prone conversions between their underlying data model and the heterogeneous formats of the incoming/outgoing data.

Contribution. Inspired by approach B), we implemented a framework for ephemeral data handling in microservices; the building blocks of software for our application areas of interest. Our framework includes a query language and an execution engine, to integrate document-oriented queries into the Jolie [146, 121] programming language. The language and our implemented framework are open-source projects[1]. Our choice on Jolie comes from the fact that Jolie programs are natively microservices [82]. Moreover, Jolie has been successfully used to build Internet-of-Things [95] and eHealth [189] architectures, as well as Process-Aware Information Systems [144], which makes our work directly applicable to our areas of interest. Finally, Jolie comes with a runtime environment that automatically translates incoming/outgoing data (XML, JSON, etc.) into the native, tree-shaped data values of the language — Jolie values for variables are always trees. By using Jolie, developers do not need to handle data conversion themselves, since it is efficiently managed by the language runtime. Essentially, by being integrated in Jolie, our framework addresses issue ⑤ by supporting variety by construction.

As main contribution of this paper, in Section 4.3, we present the formal model, called TQuery, that we developed to guide the implementation of our Jolie framework. TQuery is inspired by MQuery [41], a sound variant of the MongoDB Aggregation Framework [141]; the most popular query language for NoSQL data handling. The reason behind our formal model is twofold. On the one hand, we abstract away implementation details and reason on the overall semantics of our model — we favoured this top-down approach (from theory to practice) to avoid inconsistent/counter-intuitive query behaviours, which are instead present in the MongoDB Aggregation Framework (see [41] for details). On the other hand, the formalisation is a general reference for implementors; this motivated the balance we kept in TQuery between formal minimality and technical implementation details — e.g., while MQuery adopts a set semantics, we use a tree semantics.

As a second contribution, in Section 4.2 we present a non-trivial use case to overview the TQuery operators, by means of their Jolie programming interfaces. The use case is also the concrete

[1] https://github.com/jolie/tquery

evaluation of MQuery and, in Section 4.3, we adopt the use case as our running example to illustrate the semantics of TQuery.

4.2 A use case from online payment fraud detection

We introduce our model with the use case from online payment fraud detection. According to recent studies, the risk of online fraud remains high among various types of payment fraud [122, 17]. Currently, there exist many different approaches for detection and prevention of payment fraud, developed in academia [166, 74] and implemented in industry [11, 22]. The last one uses machine-learning techniques to predict a fraud based on previously collected data. This data is used to train a model, which, analyzing input values, can identify a possible fraud.

For our case study, we use a sample model in the form of a decision tree from [11], which has the following rules: if the amount of an online transaction is higher than twenty USD, and the card was recently used in more than two countries, we suspect a fraud. We complement this model with two special rules. First, we check the transactions selectively, choosing only those whose status is invalid or unauthorized. Second, we implement the blacklist check from [22]. Blacklists are used to signal a potential fraud and force to review the transaction more carefully. We ask the issuing bank extra details on a card used in the transaction: issuing country, card owner's address, BIN (first six numbers of a card), to check if any of them are blacklisted.

The use case in Listing 4.1 shows the Jolie microservice for fraud detection (FDM) implementing the model described above. It demonstrates the application of all TQuery operators: `match`, `unwind`, `project`, `group`, and `lookup`. In this section, we omit operator outputs and cover them in the examples of section 4.3. Listing 4.2 shows the JSON-alike sample data for transactions and card details, which we use in the case study. The data is structured in arrays (marked `[ ]`), which contain tree-shaped elements (marked `{ }`). Lines 1–9 provide the data of the initial transaction acquired from the payment service provider (PSP) microservice, lines 11–25 — the data on card involved in this transaction, and lines 27–35 — the data on other card transactions.

At lines 1–2 of listing 4.1, we evaluate the `transaction` variable: if the transaction amount is higher than 20 USD, we start checking if the data satisfies other rules of the model. At lines 3–6,

Listing 4.1: *Fraud Detection Algorithm.*

```
1  transaction;
2  if (transaction.amount > 20 ){
3      transaction.card
4      |> getCardDetails@PSP
5      |> project {country in issuingCountry, address_zip in zip, BIN,
           transaction.card in cardID}
6      |> cardDetails ;
7      checkBlackLists@FDM( cardDetails ) ( isRisky );
8
9      if( isRisky ){
10         transaction.card
11         |> getTransactionsData@PSP
12         |> unwind { M.D.L }
13         |> project {y in year ,M.m in month ,M.D.d in day, M.D.transactions
               .status in status, M.D.d.transactions.country in country}
14         |> match {status == invalid || status == authorisationRefused }
15         |> group {country by day, month, year}
16         |> project {year, month, day, country, transaction.card in cardID}
17         |> lookup {cardID == transaction.card in cardDetails}
18         |> detectFraud@FDM
19     }
20 }
```

we retrieve and shape the information about the card involved in the transaction. We use the chaining operator `|>` to pass the intermediate results of calling external services (marked with `@`) or internal `TQuery` operators. We pass the card identifier (line 3–4) to the `getCardDetails` service offered by PSP microservice. The obtained results we pass to `project` operator (line 5), which creates a new data structure with the data from the requested columns only (card's issuing country, BIN and identifier, card owner's zip number). `Project` operator also renames column names with `in` directive (e.g.`transaction.card in cardID`). Finally, `project` operator passes the result to the variable `cardDetails`.

We check if some of card details were blacklisted (line 7), calling the internal `checkBlackLists` service. In case of any risky results, we collect other suspicious card transactions and send all the data together to the `detectFraud` service for further analysis (line 10–18). We retrieve the data on other card transactions (lines 10–11) from the `getTransactionData` service at PSP microservice. At line 12, `unwind` operator flatten the structure of obtained data. `Unwind` generates a new data structure for each nested node of a root node (`year`) and its descendants separated by the dot (`M.D.transactions`). `Project` operator simplifies the obtained structure (line 13) renaming `y in year`, `M.m in month`, `M.D.d in day`, `M.D.transactions.status in status`, and `M.D.transactions.country in country`, and discarding all other columns. At

Listing 4.2: *Selective transaction data.*

```
[{
  id: "C63z9IGFR",
  object: "issuing.transaction",
  amount: 100,
  card: "7eo2Cw2lZv",
  cardholder: null,
  created: 12092919,
  ...
}]

[{
  id: "7eo2Cw2lZv",
  object: "card",
  address_city: "Odense",
  address_country: "Denmark",
  address_line1: "Campusvej",
  address_line2: 324,
  address_zip: 5200,
  brand: "Visa",
  country: "Denmark",
  exp_month: 8,
  exp_year: 2020,
  BIN: 357395,
  ...
}]

[{y:2019,M:[...,{m:8,D:[{d:29, transactions:[
{id:"3z9IGFR",time:"21:01",amount:34,currency:"DKK",status:"Authorized",
    merchant_name:"Oltrade",merchant_country:"Denmark"},
{id:"Fkl53FR",time:"23:01",amount:53,currency:"EUR",status:"Invalid",
    merchant_name:"OlutOy",merchant_country:"Finland"},
{id:"x8F83fR",time:"23:45",amount:247,currency:"EUR",
 status:"authorisationRefused",merchant_name:"OlutOy",merchant_country:"
    Finland"},...]},
{d:30, transactions:[
{id:"m34TSt6",time:"09:15",amount:79,currency:"USD",
 status:"authorisationRefused",merchant_name:"BestBuy",merchant_country:"
    USA"},
...]},...]},...]},...]
```

line 14, `match` operator filters the data to have only transactions with the stratus `invalid` or `authorisationRefused`. At line 15, `group` aggregates the countries from which the card was used, grouping them **`by`** `day`, `month` and `year`. At line 16, we `project` the data again, getting rid of the columns we have no interest in, and adding a new column for storing `cardId`, filled with `transaction.card` data. At line 17, `lookup` operator joins all obtained information on the initial and previous transactions. Finally, all the data is passed to the `detectFraud` service for further analysis.

4.3 TQuery Framework

In this section, we define the formal syntax and semantics of the operators of TQuery. We begin by defining data trees:

$$T \ni t ::= b\ \{k_i : a_i\}_i \qquad\qquad A \ni a ::= [t_1, \cdots, t_n]$$

Above, each tree $t \in T$ has two elements. First, a root value b, $b \in \texttt{sort}$, where $\texttt{sort} = \texttt{str} \cup \texttt{int} \cup \cdots \cup \{v\}$ and v is the null value. Second, a set of one-dimensional vectors, or arrays, containing sub-trees. Each array is identified by a label $k \in K$. We write arrays $a \in A$ using the standard notation $[t_1, \cdots, t_n]$. We write $k(t)$ to indicate the extraction of the array pointed by label k in t: if k is present in t we return the related array, otherwise we return the null array α, formally

$$k(\ b\ \{k_i : a_i\}_i\ ) = \begin{cases} a & \text{if } (k : a) \in \{k_i : a_i\}_i \\ \alpha & \text{otherwise} \end{cases}$$

We assume the range of arrays to run from the minimum index 1 to the maximum $\#a$, which we also use to represent the size of the array. We use the standard index notation $a[i]$ to indicate the extraction of the tree at index i in array a. If a contains an element at index i we return it, otherwise we return the null tree τ.

$$a[i] = \begin{cases} t_i & \text{if } a = [t_1, \cdots, t_n] \wedge 1 \leq i \leq n \\ \tau & \text{otherwise} \end{cases}$$

Example 4.3.1 (Data Structures)**.** To exemplify our notion of trees, we model a simplified version of the transactions data structure from Listing 4.2.

```
1 [υ{y:[2019{}],M:[υ{m:[8{}],D:[υ{d:[29{}], transactions:[υ
2 {id:"[3z9IGFR"{}],time:["21:01"{}],amount:[34{}],currency:["DKK"{}],status:
    ["Authorized"{}],merchant_name:["Oltrade"{}],merchant_country:["Denmark"
    {}]},
3 υ{id:["Fkl53FR"{}],time:["23:01"{}],amount:[53{}],currency:["EUR"{}],status
    :["Invalid"{}],merchant_name:["OlutOy"{}],merchant_country:["Finland"{}]
    },
4 υ{id:["x8F83fR"{}],time:["23:45"{}],amount:[247{}],currency:["EUR"{}],
    status:["authorisationRefused"{}],merchant_name:["OlutOy"{}],
    merchant_country:["Finland"{}]}]},
5 υ{d:[30{}], transactions:[υ{id:["m34TSt6"{}],time:["09:15"{}],amount:[79{}]
    ,currency:["USD"{}], status:["authorisationRefused"{}],merchant_name:["
    BestBuy"{}],merchant_country:["USA"{}]}]}]}]}]
```

Note that tree roots hold the values in the data structure (e.g.,2019). When root values are absent, we use the null value υ.

We define paths $p \in P$ to express tree traversal: $P \ni p ::= e\ .\ p \mid \epsilon$. Paths are concatenations of expressions e, each assumed to evaluate to a tree-label, and the sequence termination ϵ (often omitted in examples). The application of a path p to a tree t, written $[\![t]\!]^p$ returns an array that contains the sub-trees reached traversing t following p. This is aligned with the behaviour of path application in MQuery which return a set of trees. In the reminder of the paper, we write $e{\downarrow}k$ to indicate that the evaluation of expression e in a path results into the label k. Also, both here and in MQuery paths neglect array indexes: for a given path $e.p$, such that $e{\downarrow}k$, we apply the subpath p to all trees pointed by k in t. We use the standard array concatenation operator $::$ where $[t_1, \cdots, t_n] = [t_1] :: \cdots :: [t_n]$. We can finally define $[\![p]\!]^t$, which either returns an array of trees or the null array α in case the path is not applicable.

$$[\![p]\!]^t = \begin{cases} [\![p']\!]^{t_1} :: \cdots :: [\![p']\!]^{t_n} & \text{if } p = e.p' \wedge e{\downarrow}k \wedge k(t) = [t_1, \cdots, t_n] \\ [\,t\,] & \text{if } p = \epsilon \\ \alpha & \text{otherwise} \end{cases}$$

In the reminder, we also assume the following structural equivalences

$$\alpha :: \alpha \equiv \alpha \qquad \alpha :: [] \equiv [] :: \alpha \equiv [] :: [] \equiv [] \qquad \alpha :: a \equiv a :: \alpha \equiv [] :: a \equiv a :: [] \equiv a$$

Example 4.3.2. Let us see some examples of path-tree application where we assume a tree $t =$ `υ {x: [υ {z: [1 {}, 2 {} ] , y: [3 {}] } ] }`

```
1 ⟦x.ϵ⟧ᵗ ⇒ [υ{z:[1{},2{}],y:[3{}]}]
2 ⟦x.z.ϵ⟧ᵗ ⇒ [1{},2{}]
```

We first present the syntax of TQuery and then dedicate a subsection to the semantics of each operator and to the running examples that illustrate its behaviour.

As in MQuery, a TQuery query is a sequence of stages s applied on an array a: $a \triangleright s \cdots \triangleright s$. The staging operator $\triangleright$ in TQuery is similar to the Jolie chaining operator `|>`: they evaluate the expression on their left, passing its result as input to the expression at their right. We report in Figure 4.1 the syntax of TQuery, which counts five stages. The match operator μ_φ selects trees according to the criterion φ. Such criterion is either the boolean truth `true`, a condition expressing the equality of the application of path p and the array a, a condition expressing the equality of the application of path p_1 and the application of a second path p_2, the existence of a path $\exists p$, and the standard logic connectives negation $\neg$, conjunction $\wedge$, and disjunction $\vee$. The unwind operator ω_p flattens an array reached through a path p and outputs a tree for each element of the array. The project operator π_Π modifies trees by projecting away paths, renaming paths, or introducing new paths, as described in the sequence of elements in Π, which are either a path p or a value definition d inserted into a path p. Value definitions are either: a boolean value b (`true` or `false`), the application of a path p, an array of value definitions, a criterion φ or the ternary expression, which, depending the satisfiability of criterion φ selects either value definition d_1 or d_2. The group operator $\gamma_{\Gamma,\Gamma'}$ groups trees according to a grouping condition Γ and aggregates values of interest according to Γ'. Both Γ and Γ' are sequences of elements of the form $p \rangle p'$ where p is a path in the input trees, and p' a path in the output trees. The lookup operator $\lambda_{q=a.r\rangle p}$ joins input trees with trees in an external array a. The trees to be joined are found by matching those input trees whose array found applying path q equals the

$$
\begin{array}{lcl}
s & ::= & \mu_\varphi \mid \omega_p \mid \pi_\Pi \mid \gamma_{\Gamma:\Gamma'} \mid \lambda_{q=a.r\rangle p} \\
\varphi & ::= & \mathtt{true} \mid p = a \mid p_1 = p_2 \mid \exists p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
\Pi & ::= & p \mid d\rangle p \mid p,\Pi \mid d\rangle p,\Pi \\
d & ::= & b \mid p \mid [d_1,\cdots,d_n] \mid \varphi \mid \varphi?d_1 : d_2 \\
\Gamma & ::= & p\rangle p' \mid p\rangle p',\Gamma
\end{array}
$$

Figure 4.1: *Syntax of the TQuery*

ones found applying path r to the trees of the external array a. The matching trees from a are stored in the matching input trees under path p.

4.3.1 Unwind

To define the semantics of the unwind operator ω, we introduce the unwind expansion operator $E(t,a)^k$ (read "unwind t on a under k"). Informally $E(t,a)^k$ returns an array of trees with cardinality $\#a$ where each element has the shape of t except that label k is associated index-wise with the corresponding element in a. Formally, given a tree t, an array a, and a key k:

$$
E(t,a)^k = \begin{cases}
\left[b\Big(\big(\{k_i : a_i\}_i \setminus \{k : k(t)\}\big) \cup \{k : [t']\}\Big)\right] :: E(t,a')^k & \text{if } \begin{array}{l} a = [t'] :: a' \\ \wedge\, t = b\{k_i : a_i\}_i \end{array} \\
[\,] & \text{otherwise}
\end{cases}
$$

Then, the formal definition of $a \triangleright \omega_p$ is

$$
a \triangleright \omega_p \begin{cases}
E(t, [\![k.\epsilon]\!]^t \triangleright \omega_{p'})^k :: a' \triangleright \omega_p & \text{if } p = e.p' \wedge e{\downarrow}k \wedge a = [t] :: a' \\
a & \text{if } p = \epsilon \\
[] & \text{otherwise}
\end{cases}
$$

We define the unwind operator inductively over both a and p. The induction over a results in the application of the unwind expansion operator E over all elements of a. The induction over p splits p in the current key k and the continuation p'. Key k is used to retrieve the array in the current element of a, i.e., $[\![k.\epsilon]\!]^t$, on which we apply ω'_p to continue the unwind application until we reach the termination with $p = \epsilon$.

Example 4.3.3. We report the execution of the unwind operator at line 12 of Listing 4.1.

The unwind operator unfolds the given input array wrt a given path p in two directions. The first is breadth, where we apply the unwind expansion operator $E(t, a')^k$, over all input trees t and wrt the first node k in the path p. The second direction is depth, and defines the content of array a' in $E(t, a')^k$, which is found by recursively applying the unwind operator wrt to the remaining path nodes in p (k excluded) over the arrays pointed by node k in each t.

Let a be the transactions data-structure at lines 27–35 of Example 4.3.1, such that $a = [t_{2019}, t_{2018}, ...]$ where e.g., t_{2019} is that t in a such that $[\![y]\!]^t =$ `[2019{}]`. The concatenation below is the first level of depth unfolding, i.e., for node `M` of unwind $\omega_{\texttt{M.D.transactions}}$.

$$E(t_{2019}, [\![\texttt{M}.\epsilon]\!]^{t_{2019}} \triangleright \omega_{\texttt{D.transactions}})^{\texttt{M}} :: E(t_{2018}, [\![\texttt{M}.\epsilon]\!]^{t_{2018}} \triangleright \omega_{\texttt{D.transactions}})^{\texttt{M}} :: \ldots$$

To conclude this example, we show the execution of the unwind expansion operator

$$E(t_{29}, [\![\texttt{transactions}.\epsilon]\!]^{t_{29}})^{\texttt{transactions}}$$

of the terminal node `transactions` in path p, relative to the transaction records recorded within a day, represented by tree t_{29}, i.e., where $[\![t_{29}]\!]^d =$ `[29{}]`.

```
E(t29, [[transactions.ε]]^t30)^transactions ⇒
[υ (({d:[29{}],transactions:[...]}\{transactions:[...]})
    ∪{transactions:[υ{id:["3z9IGFR"{}],time:["21:01"{}],... }]})] ::
[υ (({d:[29{}],L:[...]}\{transactions:[...]})
    ∪{transactions:[υ{id:["Fkl53FR"{}],time:["23:01"{}],... }]})] ::
[υ (({d:[29{}],L:[...]}\{transactions:[...]})
    ∪{transactions:[υ{id:["x8F83fR"{}],time:["23:45"{}],... }]})] :: ...
```

Above, for each element of the array pointed by `transactions`, we create a new structure `[υ (({d:[29{}],transactions:[...]})]` where we replace the original array associated with the key `transactions` with a new array containing only that element.

The final result of the unwind operator has the shape:

```
[υ{y:[2019{}],M:[{m:[8{}],D:[{d:[29{}],
   transactions:[υ{id:["3z9IGFR"{}],time:["21:01"{}],amount:[34{}],
     currency:["DKK"{}],status:["Authorized"{}],
     merchant_name:["Oltrade"{}],merchant_country:["Denmark"{}]}]}]}]},
```

```
υ{y:[2019{}],M:[υ{m:[8{}],D:[υ{d:[29{}],
  transactions:[υ{id:["Fkl53FR"{}],time:["23:01"{}],amount:[53{}],
    currency:["EUR"{}],status:["Invalid"{}],
    merchant_name:["OlutOy"{}],merchant_country:["Finland"{}]}]}]}]},
υ{y:[2019{}],M:[{m:[8{}],D:[υ{d:[29{}],
  transactions:[υ{id:["x8F83fR"{}],time:["23:45"{}],amount:[247{}],
    currency:["EUR"{}],status:["authorisationRefused"{}],
    merchant_name:["OlutOy"{}],merchant_country:["Finland"{}]}]}]}]},
```

4.3.2 Project

We start by defining some auxiliary operators used in the definition of the project. Auxiliary operators $\pi_p(a)$ and $\pi_p(t)$ formalise the application of a branch-selection over a path p. Then, the auxiliary operator **eval**(d,t) returns the array resulting from the evaluation of a definition d over a tree t. Finally, we define the projection of a value (definition) d into a path p over a tree t i.e., $\pi_{d\rangle p}(t)$. The projection for a path p over an array a results in an array where we project p over all the elements (trees) of a.

$$\pi_p([t_1,\cdots,t_n]) = [\,\pi_p(t_1),\cdots,\pi_p(t_n)\,]$$

The projection for a path p over a tree t implements the actual semantics of branch-selection, where, given a path $e.p'$, $e{\downarrow}k$, we remove all the branches k_i in $t = b\{k_i : a_i\}$, keeping only k (if $k \in \{k_i\}$) and continue to apply the projection for the continuation p' over the (array of) sub-trees under k in t (i.e., $[\![k.\epsilon]\!]^t$).

$$\pi_p(t) = \begin{cases} \upsilon\,\{\,k : \pi_{p'}([\![k.\epsilon]\!]^t)\,\} & \text{if } [\![p]\!]^t \neq \alpha \wedge p = e.p' \wedge t = b\,\{k_i : a_i\}_i \wedge e{\downarrow}k \\ t & \text{if } p = \epsilon \\ \tau & \text{otherwise} \end{cases}$$

The operator $\mathbf{eval}(d, t)$ evaluates the value definition d over the tree t and returns an array containing the result of the evaluation.

$$\mathbf{eval}(d,t) = \begin{cases} [\, d\,\{\}\,] & \text{if } d \in V \\ [\, t \models \varphi\,\{\}\,] & \text{if } d \in \varphi \\ [\![d]\!]^t & \text{if } d \in P \\ \mathbf{eval}(d,t) :: \mathbf{eval}(d',t) & \text{if } d = [d] :: d' \\ \mathbf{eval}(d',t) & \text{if } d = \varphi ? d_{\mathtt{true}} : d_{\mathtt{false}} \wedge d' = d_{t \models \varphi} \\ \alpha & \text{otherwise} \end{cases}$$

Then, the application of the projection of a value definition d on a path p, i.e., $\pi_{d \rangle p}(t)$ returns a tree where under path p is inserted the **eval**uation of d over t.

$$\pi_{d \rangle p}(t) = \begin{cases} \upsilon\left\{k : [\, \pi_{d \rangle p'}(t)\,]\right\} & \text{if } p = e.p' \wedge e{\downarrow}k \wedge \mathbf{eval}(d,t) \neq \alpha \\ \upsilon\left\{k : \mathbf{eval}(d,t)\right\} & \text{if } p = e.\epsilon \ \wedge e{\downarrow}k \wedge \mathbf{eval}(d,t) \neq \alpha \\ \tau & \text{otherwise} \end{cases}$$

Before formalising the projection, we define the auxiliary tree-merge operator $t \oplus t'$, used to merge the result of a sequence of projections Π.

$$([t] :: a) \oplus ([t'] :: a') = [t \oplus t'] :: a \oplus a' \qquad a \oplus [] = [] \oplus a = a \oplus \alpha = \alpha \oplus a = a$$

$$b\,\{k_i : a_i\}_i \oplus b'\,\{k_j : a_j\}_j = \tau \quad \text{if } b \neq b' \qquad t \oplus \tau = t$$

$$t \oplus t' = b\,\{\, k_h : k_h(t) \oplus k_h(t')\,\}_{h \in I \cup J} \quad \text{if } t = b\,\{\, k_i : a_i\,\}_{i \in I} \wedge t' = b\,\{\, k_j : a_j\,\}_{j \in J}$$

To conclude, first we define the application of the projection to a tree t, i.e., $t \triangleright \pi_{\Pi}$, which merges ($\oplus$) into a single tree the result of the applications of projections Π over t

$$\pi_{\Pi}(t) = \begin{cases} \pi_{p}(t) \oplus (t \triangleright \pi_{\Pi'}) & \text{if } \Pi = p, \Pi' \\ \pi_{d\rangle p}(t) \oplus (t \triangleright \pi_{\Pi'}) & \text{if } \Pi = d\rangle p, \Pi' \\ \pi_{p}(t) & \text{if } \Pi = p \\ \pi_{d\rangle p}(t) & \text{if } \Pi = d\rangle p \end{cases}$$

and finally, we define the application of the projection π_{Π} to an array a, i.e., $a \triangleright \pi_{\Pi}$, which corresponds to the application of the projection to all the elements of a.

$$[\, t_1, \cdots, t_n \,] \triangleright \pi_{\Pi} = [\, \pi_{\Pi}(t_1), \cdots, \pi_{\Pi}(t_n) \,]$$

Example 4.3.4. We report the execution of the `project` at line 5 of Listing 4.1. Let a be the array corresponding to the data structure at lines 11–25 of listing 4.2, and let t be the tree from a

```
a = [υ{id: ["7eo2Cw2lZv"{}],object:["card"{}],address_city:["Odense"{}],
       address_country:["Denmark"{}],address_line1:["Campusvej"{}],
       address_line2:[324{}],address_zip:[5200{}],brand:["Visa"{}],
       country:["Denmark"{}],exp_month:[8{}],exp_year:[2020{}],
       BIN:[357395{}]
}]
```

$[t] \triangleright \pi_{\texttt{country}\rangle\texttt{issuingCountry},\ \texttt{address\_zip}\rangle\texttt{zip},\ \texttt{BIN},\ \texttt{transaction.card}\rangle\texttt{cardID}} \Rightarrow$

$[\pi_{\texttt{country}\rangle\texttt{issuingCountry},\ \texttt{address\_zip}\rangle\texttt{zip},\ \texttt{BIN},\texttt{transaction.card}\rangle\texttt{cardID}}(t)] \Rightarrow$

$[\pi_{\texttt{country}\rangle\texttt{issuingCountry}}(t) \oplus \pi_{\texttt{address\_zip}\rangle\texttt{zip}}(t) \oplus \pi_{\texttt{BIN}}(t) \oplus \pi_{\texttt{transaction.card}\rangle\texttt{cardID}}(t)] \Rightarrow$

$\upsilon\{\texttt{issuingCountry}:\pi_{[\text{"Denmark"}\{\}]}(t)\} \oplus \upsilon\{\texttt{zip}:\pi_{[\text{“5200”}\{\}]\rangle\epsilon}(t)\} \oplus \upsilon\{\texttt{BIN}:\pi_{[\text{“357395”}\{\}]\rangle\epsilon}(t)\} \oplus \upsilon\{\texttt{transaction.card}:\pi_{[\text{“7eo2Cw2lZv”}\{\}]\rangle\epsilon}(t)\} =$

$\upsilon\{\texttt{issuingCountry}:\textbf{eval}([\text{"Denmark"}\{\}],(t)\} \oplus \upsilon\{\texttt{zip}:\textbf{eval}([\text{"5200"}\{\}],t)\} \oplus \upsilon\{\texttt{BIN}:\textbf{eval}([\text{"357395"}\{\}],t)\} \oplus \upsilon\{\texttt{transaction.card}:\textbf{eval}([\text{"7eo2Cw2lZv"}\{\}],t)\}$

The final result of the projection has the shape

```
[υ{issuingCountry:["Denmark"{}],zip:[5200{}],BIN:[357395{}],
           cardId:["7eo2Cw2lZv"{}]}]
```

Example 4.3.5. We report the execution of the `project` at line 13 of Listing 4.1. Let α be the array at the end of Example 4.3.3, and let t^1_{2019}, t^2_{2019}, ... be the trees in α such that t^1_{2019} is the first tree in α relative to `year` 2019, t^2_{2019} the second, and so on.

$$[t^1_{2019}, t^2_{2019}, \ldots] \triangleright \pi_{\text{y}\rangle\text{year},\ \text{M.m}\rangle\text{month},\ \text{M.D.d}\rangle\text{day},\ \text{M.D.L.q}\rangle\text{quality}} \Rightarrow$$
$$[\pi_{\text{y}\rangle\text{year},\ \text{M.m}\rangle\text{month},\ \text{M.D.d}\rangle\text{day},\ \text{M.D.L.q}\rangle\text{quality}}(t^1_{2019}), \pi_{\text{y}\rangle\text{year},\ \text{M.m}\rangle\text{month},\ \text{M.D.d}\rangle\text{day},\ \text{M.D.L.q}\rangle\text{quality}}(t^2_{2019}), \ldots]$$

We continue showing the projection of the first element in α, t^1_{2019} (the projection on the other elements follows the same structure)

$$\pi_{\text{y}\rangle\text{year},\ \text{M.m}\rangle\text{month},\ \text{M.D.d}\rangle\text{day},\ \text{M.D.transactions.status}\rangle\text{status},\ \text{M.D.transactions.country}\rangle\text{country}}(t^1_{2019}) \Rightarrow$$
$$\pi_{\text{y}\rangle\text{year}}(t^1_{2019}) \oplus \pi_{\text{M.m}\rangle\text{month}}(t^1_{2019}) \oplus \pi_{\text{M.D.d}\rangle\text{day}}(t^1_{2019}) \oplus \pi_{\text{M.D.transactions.status}\rangle\text{status}}(t^1_{2019}) \oplus$$
$$\pi_{\text{M.D.transactions.country}\rangle\text{country}}(t^1_{2019})$$

Finally, we show the unfolding of the first two projections from the left, above, i.e., those for $\text{y}\rangle\texttt{year}$ and for $\texttt{M.m}\rangle\texttt{month}$, and their merge $\oplus$ (the remaining ones unfold similarly).

$$\pi_{\text{y}\rangle\text{year}}(t^1_{2019}) \oplus \pi_{\text{M.m}\rangle\text{month}}(t^1_{2019}) \Rightarrow$$
$$\upsilon\ \{\texttt{year}\ :\ \pi_{\text{y}\rangle\epsilon}(t^1_{2019})\ \} \oplus \upsilon\ \{\texttt{month}\ :\ \pi_{\text{M.m}\rangle\epsilon}(t^1_{2019})\ \}$$
$$= \upsilon\ \{\texttt{year}\ :\ \textbf{eval}(\text{y}, t^1_{2019})\ \} \oplus \upsilon\ \{\texttt{month}\ :\ \textbf{eval}(\text{M.m}, t^1_{2019})\ \}$$
$$= \upsilon\ \{\texttt{year}\ :\ [\![\text{y}]\!]^{t^1_{2019}}\ \} \oplus \upsilon\ \{\texttt{month}\ :[\![\text{M.m}]\!]^{t^1_{2019}}\ \}$$
$$= \upsilon\ \{\texttt{year}\ :\ [2019\{\}]\ \} \oplus \upsilon\ \{\texttt{month}\ :[8\{\}]\ \}$$
$$= \upsilon\ \{\texttt{year}\ :\ [2019\{\}],\ \texttt{month}\ :\ [8\{\}]\ \}$$

The result of the projection has the shape

```
[υ{year:[2019{}],month[8{}],day{29[]},status:["Authorized"{}],
    country:["Denmark"{}]},
υ{year:[2019{}],month[8{}],day{29[]},status:["Invalid"{}],
    country:["Finland"{}]},
υ{year:[2019{}],month[8{}],day{29[]},status:["authorisationRefused"{}],
    country:["Finland"{}]},
υ{year:[2019{}],month[8{}],day{30[]},status:["authorisationRefused"{}],
    country:["USA"{}]},
```

4.3.3 Match

When applied to an array α, match μ_φ returns those elements in α that satisfy φ. If there is no element in α that satisfies φ, μ_φ returns an array with no elements (different from α). Below,

we mark $t \models \varphi$ the satisfiability of criterion φ by a tree t.

$$\alpha \triangleright \mu_\varphi = [\,] \qquad [t] :: a \triangleright \mu_\varphi = \begin{cases} [t] :: (a \triangleright \mu_\varphi) & \text{if } t \models \varphi \\ a \triangleright \mu_\varphi & \text{if } \#a > 0 \\ [\,] & \text{otherwise} \end{cases}$$

$$t \models \varphi \text{ holds iff } \begin{cases} \varphi = \texttt{true} \\ \varphi = (\exists p) \wedge [\![p]\!]^t \neq \alpha \\ \varphi = (p = a) \wedge [\![p]\!]^t = a \\ \varphi = (p_1 = p_2) \wedge t \models \big((p_1 = a) \wedge (p_2 = a)\big) \\ \varphi = (\neg \varphi') \wedge t \not\models \varphi' \\ \varphi = (\varphi_1 \wedge \varphi_2) \wedge (t \models \varphi_1 \wedge t \models \varphi_2) \\ \varphi = (\varphi_1 \vee \varphi_2) \wedge (t \models \varphi_1 \vee t \models \varphi_2) \end{cases}$$

Above, criterion $\varphi = (p_1 = p_2)$ is satisfied both when the application of the two paths to the input tree t return the same array a as well as when both paths do not exist in t, i.e., their application coincide on α.

Example 4.3.6. We report below the execution of the `match` operator at line 14 of Listing 4.1. In the example, array a corresponds to the following data structure:

```
1 [υ{year:[2019{}],month[8{}],day{29[]},status:["Authorized"{}],
2     country:["Denmark"{}]},
3 υ{year:[2019{}],month[8{}],day{29[]},status:["Invalid"{}],
4     country:["Finland"{}]},
5 υ{year:[2019{}],month[8{}],day{29[]},status:["authorisationRefused"{}],
6     country:["Finland"{}]},
7 υ{year:[2019{}],month[8{}],day{30[]},status:["authorisationRefused"{}],
8     country:["USA"{}]},
```

. First we formalise in TQuery the `match` operator at line 14: $a \triangleright \mu_\varphi$ where

$$\varphi = \texttt{status} == \texttt{["authorisationRefused"\{\}]} \vee \texttt{status} == \texttt{["Invalid"\{\}]}$$

The match evaluates all trees inside a, below we just show that evaluation for $a[1]$

```
a[1] = υ{year:[2019{}],month[8{}],day{29[]},status:["Authorized"{}],
        country:["Denmark"{}]}
```

we verify if one of the sub-conditions $a[1] \models$ `status == ["authorisationRefused"{}]` or $a[1] \models$ `status == ["Invalid"{}]` hold. Each condition is evaluated by applying path `status` on $a[1]$ and by verifying if the equality with the considered array, e.g., `["Invalid"{}]`, holds. As a result, we obtain the input array a filtered from the trees that do not correspond to the dates in the criterion.

4.3.4 Group

The group operator takes as parameters two sequences of paths, separated by a semicolon, i.e., $q_1\rangle p_1, \cdots, q_n\rangle p_n : s_1\rangle r_1, \cdots, s_m\rangle r_m$. The first sequence of paths, ranged $[1, n]$, is called aggregation set, while the second sequence, ranged $[1, m]$, is called grouping set. Intuitively, the group operator first groups together the trees in a which have the maximal number of paths $s_1, \cdots, s_m$ in the grouping set whose values coincide. The values in $s_1, \cdots, s_m$ are projected in the corresponding paths $r_1, \cdots, r_m$. Once the trees are grouped, the operator aggregates all the different values, without duplicates, found in paths $q_1, \cdots, q_n$ from the aggregation set, projecting them into the corresponding paths $p_1, \cdots, p_n$. We start the definition of the grouping operator by expanding its application to an array a. In the expansion below, on the right, we use the series-concatenation operator $\vdots$ and the set H, element of the power set $2^{[1,n]}$, to range over all possible combinations of paths in the grouping set. Namely, the expansion corresponds to the concatenation of all the arrays resulting from the application of the group operator on a subset (including the empty and the whole set) of paths in the grouping set.

$$\gamma_{q_1\rangle p_1, \cdots, q_n\rangle p_n : s_1\rangle r_1, \cdots, s_m\rangle r_m} \triangleright a = \vdots_{\forall H \in 2^{[1,m]}} \gamma^{H}_{q_1\rangle p_1, \cdots, q_n\rangle p_n : s_1\rangle r_1, \cdots, s_m\rangle r_m}(a)$$

In the definition of the expansion, we mark $\{\{a\}\}$ the casting of an array a to a set (i.e., we keep only unique elements in a and lose their relative order). Each $\gamma^{H}_{q_1\rangle p_1, \cdots, q_n\rangle p_n : s_1\rangle r_1, \cdots, s_m\rangle r_m}(a)$ returns an array that contains those trees in a that correspond to the grouping illustrated above.

Formally:

$$\gamma^{H}_{q_1\rangle p_1,\cdots,q_n\rangle p_n:s_1\rangle r_1,\cdots,s_m\rangle r_m}(a) =$$

$$\begin{cases} \vdots\\ \forall a' \end{cases}\left[\bigoplus_{i=1}^{n}\pi_{\chi,\theta_i}(\tau)\right] \quad \text{if} \quad \begin{array}{l} h \in H \wedge a'[h] \in \left\{[\![s_h]\!]^t \mid t \in \{\!\{a\}\!\} \wedge [\![s_h]\!]^t \neq \alpha\right\} \\ \wedge\, \chi = (a'[h]\rangle H, [r_1,\cdots,r_n]) \\ \wedge\, \theta_i = \begin{cases} \vdots\\ \forall t_i \end{cases}[\![q_i]\!]^{t_i}\rangle p_i \wedge t_i \in \{\{a \triangleright \mu_{\psi_i}\}\} \supset \varnothing \\ \wedge\, \psi_i = \exists q_i \wedge \neg \bigvee_{j \notin H} \exists s_j \wedge \bigwedge_h \left((s_h = a'[h]) \wedge \exists s_h\right) \end{array}$$

$$[\,] \qquad \text{otherwise}$$

When applied over a set H, $h \in H$, γ considers all combinations of values identified by paths s_h in the trees in a. In the formula above, we use the array a' to refer to those combinations of values. In the definition, we impose that, for each element in a' in a position h, there must be at least one tree in a that has a non-null ($\neq \alpha$) array under path s_h. Hence, for each combination a' of values in a, γ builds a tree that i) contains under paths r_h the value $a'[h]$ (as encoded in the projection query χ and from the definition of the operator $a'[h]\rangle H, a$, defined below) and ii) contains under paths p_i, $1 \leq i \leq n$, the array containing all the values found under the correspondent path q_i in all trees in a that match the same combination element-path in a' (as encoded in θ_i). The grouping is valid (as encoded in ψ_i) only if we can find (i.e., match μ) trees in a where i) we have a non-empty value for q_i, ii) there are no paths s_j that are excluded in H, and iii) for all paths considered in H, the value found under path s_h corresponds to the value in the considered combination $a'[h]$. If the previous conditions are not met, γ returns an empty array $[\,]$.

We conclude defining the operator $a'[h]\rangle H, a$, used above to unfold the set of aggregation paths and the related values contained in H, e.g., let $H = \{1,3,5\}$ then $a'[h]\rangle H, a = a'[1]\rangle p_1, a'[3]\rangle p_3, a'[5]\rangle p_5$. Its meaning is that, for each path $p_\bullet$, we project in it the value correspondent to $a'[\bullet]$. Formally

$$a'[h]\rangle H, a = \begin{cases} a'[j]\rangle a[j], (a'[h]\rangle (H \setminus \{j\}), a) & \text{if } |H| > 1 \wedge j \in H \\ a'[j]\rangle a[j] & \text{if } |H| = 1 \wedge j \in H \\ \epsilon & \text{otherwise} \end{cases}$$

Note that for case $\gamma^{\varnothing}_{\dots}$ (i.e., for $H = \varnothing$), $a'[h]\rangle H, a$ returns the empty path ϵ, which has no effect (i.e., it projects the input tree) in the projection π_χ in the definition of γ. Hence, the resulting tree from grouping over $\varnothing$ will just include (and project over $p_1, \cdots, p_n$) those trees in a that do not include any value reachable by paths $s_1, \cdots, s_m$ (as indicated by expression $\neg \bigvee_{j \notin H} \exists s_j$ in ψ_i).

Like in MQuery and MongoDB, we allow the omission of paths $p_1, \cdots, p_n$ and $r_1, \cdots, r_n$ in $\Gamma : \Gamma'$. However, we interpret this omission differently wrt MQuery. There, the values obtained from q_is with missing p_is (resp., s_i with missing r_i) are stored within a default path `_id`. Here, we intend the omission as an indication of the fact that the user wants to preserve the structure of q_i (resp., s_i) captured by the structural equivalence below.

$$\gamma_{q_1,\cdots,q_n:s_1,\cdots,s_m} \equiv \gamma_{q_1\rangle q_1,\cdots,q_n\rangle q_n:s_1\rangle s_1,\cdots,s_m\rangle s_m}$$

Example 4.3.7. We report the execution of the `group` operator at line 14 of Listing 4.1. Let a be the result of the projection Example 4.3.5, with the exception that a has been filtered by the `match` at line 14 in Listing 4.1, `status` column has been removed, and a contains only the transaction records for `days` `29` and `30` of `month` `8` and `year` `2019`.

$$a \triangleright \gamma_{\texttt{country} \,:\, \texttt{day},\texttt{month},\texttt{year}} \equiv a \triangleright \gamma_{\texttt{country}\rangle\texttt{country} \,:\, \texttt{day}\rangle\texttt{day},\texttt{month}\rangle\texttt{month},\texttt{year}\rangle\texttt{year}}$$

$$\Rightarrow \bigvee_{\forall H \in 2^{[1,1]}} \gamma^{H}_{\texttt{country}\rangle\texttt{country} \,:\, \texttt{day}\rangle\texttt{day},\texttt{month}\rangle\texttt{month},\texttt{year}\rangle\texttt{year}}(a)$$

$$= \left[\gamma^{\varnothing}_{\texttt{country}\rangle\texttt{country} \,:\, \texttt{day}\rangle\texttt{day},\texttt{month}\rangle\texttt{month},\texttt{year}\rangle\texttt{year}}(a)\right] \quad \text{this to equals } [\,]$$

$$:: \left[\gamma^{\{1\}}_{\texttt{country}\rangle\texttt{country} \,:\, \texttt{day}\rangle\texttt{day},\texttt{month}\rangle\texttt{month},\texttt{year}\rangle\texttt{year}}(a)\right] \quad \text{since } \psi_i \text{ is always false in } a$$

$$= [\,] :: \left[\pi_{\texttt{[30 \{\}]}\rangle\texttt{day},\texttt{[8 \{\}]}\rangle\texttt{month},\texttt{[2019 \{\}]}\rangle\texttt{year}}(\tau) \sqcup \pi_{\texttt{country}\rangle\texttt{["USA"\{\}]}}(\tau)\right]$$

$$:: \left[\pi_{\texttt{[29 \{\}]}\rangle\texttt{day},\texttt{[8 \{\}]}\rangle\texttt{month},\texttt{[2019 \{\}]}\rangle\texttt{year}}(\tau) \sqcup \pi_{\texttt{country}\rangle\texttt{["Finland"\{\}]}}(\tau)\right]$$

$$= [\upsilon\{\ \texttt{day: [30\{\}],month:[8\{\}],year:[2019\{\}] ,country:["USA"\{\}]},}$$

$$\upsilon\{\ \texttt{day: [29\{\}],month:[8\{\}],year:[2019\{\}] ,country:["Finland"\{\}]}]$$

4.3.5 Lookup

Informally, the lookup operator joins two arrays, a source a and an adjunct a', wrt a destination path p and two source paths q and r. Result of the lookup is a new array that has the shape of the source array a but where each of its elements t has under path p those elements in the adjunct array a' whose values under path r equal the values found in t under path q. Formally

$$a \triangleright \lambda_{q=a'.r\rangle p} = [\,\pi_{\epsilon,\beta_1}(a[1])\,] :: \cdots :: [\,\pi_{\epsilon,\beta_n}(a[n])\,] \text{ s.t. } \begin{cases} \beta_i = (a' \triangleright \mu_{r=a''})\rangle p \\ \wedge\ a'' = [\![q]\!]^{a[i]} \\ \wedge\ 1 \leq i \leq n \end{cases}$$

Above, the lookup operator λ takes as parameters three paths p, q, and r and an array of trees a'. When applied to an array of trees $a = [t_1, \cdots, t_n]$, it returns a (i.e., all of its elements, as retuned by the projection π under the first parameter ϵ) where each of its elements has under path p an array of trees obtained from applying the match ($\mu_{r=a''}$) in expression β_i, i.e., following the definition of π, the projection under ϵ is merged with the result of the projection under β_i. For each element $a[i]$ ($1 \leq i \leq n$), β_i matches those trees in a' for which either i) there is a path r and the array reached under r equals the array found under $[\![q]\!]^{a[i]}$ or ii) there exist no path r (i.e., its application returns the null array α) and also q does not exist in t_i (i.e., $[\![q]\!]^{a[i]} = \alpha$).

Example 4.3.8. We report the execution of the `lookup` at line 17 of Listing 4.1

$$a \triangleright \lambda_{\texttt{patient\_id}=a'.\texttt{patient\_id}\rangle \texttt{temps}}$$

where a corresponds to the resulting array from the application of the `project` operator at line 16 of Listing 4.1, which has the shape

```
1 a =
2 [υ{day:[30{}],month:[8{}],year[2019{}],country:["USA"{}],
3     cardID:["7eo2Cw2lZv"{}]},
4  υ{day:[29{}],month:[8{}],year[2019{}],country:["Finland"{}],
5     cardID:["7eo2Cw2lZv"{}]}]
```

and where a' corresponds to the array of card details that results from the application of the `project` at line 5 of Listing 4.1, as shown at the bottom of Example 4.3.4. Then, unfolding

the execution of the lookup, we obtain the concatenation of the results of two projections, on the only two elements in α. The first corresponds to the projection on ϵ, β_1 while the second corresponds to the projection on ϵ, β_2 where

$$\beta_1 = \beta_2 = \alpha' \triangleright \mu_{\texttt{cardID=["xxx" \{\}]}} \langle \texttt{cardDetails}$$

Below, sub-node `cardDetails` contains the whole array α', since all its elements match `cardID`.

```
[π_{ε,β1}(α[1])::π_{ε,β2}(α[2])]
= [υ{day:[30{}],month:[8{}],year[2019{}],country:["USA"{}],
     cardID:["7eo2Cw2lZv"{}],
     cardDetails:
        [υ{issuingCountry:["Denmark"{}],zip:[5200{}],BIN:[357395{}],
           cardId:["7eo2Cw2lZv"{}]}]},
   υ{day:[29{}],month:[8{}],year[2019{}],country:["Finland"{}],
     cardID:["7eo2Cw2lZv"{}],
     cardDetails:
        [υ{issuingCountry:["Denmark"{}],zip:[5200{}],BIN:[357395{}],
           cardId:["7eo2Cw2lZv"{}]}]}]
```

4.4 Related Work and Conclusion

In this chapter, we focus on ephemeral data handling and contrast DBMS-based solutions wrt to integrated query engines within a given application memory. We indicate issues that make unfit DBMS-based solutions in ephemeral data-handling scenarios and propose a formal model, called TQuery, to express document-based queries over common (JSON, XML, ...), tree-shaped data structures.

TQuery instantiates MQuery [41], a sound variant of the Aggregation Framework [141] used in MongoDB, one of the main NoSQL DBMSes for document-oriented queries.

We implemented TQuery in Jolie, a language to program native microservices, the building blocks of modern systems where ephemeral data handling scenarios are becoming more and more common, like in Internet-of-Things, eHealth, and Edge Computing architectures. Jolie offers variety-by-construction, i.e., the language runtime automatically and efficiently handles data

conversion, and all Jolie variables are trees. These factors allowed us to separate input/output data-formats from the data-handling logic, hence providing programmers with a single, consistent interface to use TQuery on any data-format supported by Jolie.

In our treatment, we presented a non-trivial use case from online payments fraud detection, which provide a concrete evaluation of both TQuery and MQuery, while also serving as a running example to illustrate the behaviour of the TQuery operators.

Regarding related work, we focus on NoSQL systems, which either target documents, key/value, and graphs. The NoSQL systems closest to ours are the MongoDB [118] Aggregation Framework, and the CouchDB [70] query language which handle JSON-like documents using the JavaScript language and REST APIs. ArangoDB [28] is a native multi-model engine for nested structures that come with its own query language, namely ArangoDB Query Language. Redis [170] is an in-memory multi-data-structure store system, that supports string, hashes, lists, and sets, however it lacks support for tree-shaped data. We conclude the list of external DB solutions with Google Big Table [61] and Apache HBase [100] that are NoSQL DB engines used in big data scenarios, addressing scalability issues, and thus specifically tailored for distributed computing. As argued in the introduction, all these systems are application-external query execution engine and therefore unfit for ephemeral data-handling scenarios.

There are solutions that integrate linguistic abstractions to query data within the memory of an application. One category is represented by Object-relation Mapping (ORM) frameworks [91]. However, ORMs rely on some DBMS, as they map objects used in the application to entities in the DBMS for persistence. Similarly, Opaleye [85] is a Haskell library providing a DSL generating PostgreSQL. Thus, while being integrated within the application programming tools and executing in-memory, in ephemeral data-handling scenarios, ORMs are affected by the same issues of DBMS systems. Another solution is LevelDB [117], which provides both a on-disk and in-memory storage library for C++, Python, and Javascript, inspired by Big Table and developed by Google, however it is limited to key-value data structures and does not support natively tree-shaped data. As cited in the introduction, a solution close to ours is LINQ [136], which provides query operators targeting both SQL tables and XML nested structures with *.NET* query operators. Similarly, CQEngine [97] provides a library for querying Java collections with

SQL-like operators. Both solutions do not provide automatic data-format conversion, as our implementation of TQuery in Jolie.

We are currently empirically evaluating the performance of our implementation of TQuery in application scenarios with ephemeral data handling (Internet-of-Things, eHealth, Edge Computing). The next step would be to use those scenarios to conduct a study comparing our solution wrt other proposals among both DBMS and in-memory engines, evaluating their impact on performance and the development process. Finally, on the one hand, we can support new data formats in Jolie, which makes them automatically available to our TQuery implementation. On the other hand, expanding the set of available operators in TQuery would allow programmers to express more complex queries over any data format supported by Jolie.

Chapter 5

Conclusion

In this thesis, we have looked at the microservices architectural style from different angles. We have explored its evolution, traits, problems, and solutions for effective communication and data handling. In this chapter, we give some conclusions and directions for future work.

Microservices, systematically

One of the goals of our survey on microservices was to create a guidebook, combining versatile information on different aspects of this architecture or at least to offer some starting points for looking at them. However, trying to embrace the constantly growing universe of microservices is an ambitious task requiring regular revision of the current state of the art. Adding more surveys, capable of systemizing the information, would help to navigate in various publications on microservices whose number has been increased recently both in academia and industry due to the popularity of the pattern. One of the possible cases could be a continuation of our research of the microservices quality model, which at the moment lacks comprehensiveness. A more systematic approach could be covering consistently one of the acknowledged software quality models (e.g ISO/IEC 25010 [12]), investigating possible trade-offs among quality attributes, new architectural patterns and guidelines aimed at helping with practical implementation. Similar work could be done e.g. for security patterns, services monitoring, ecosystem, etc.

Enhancing choreography extraction

By implementing the algorithm of choreographies extraction, we have approached the problem of efficient computing of coordination model for a set of processes. The algorithm generates a choreographic model of process coordination based on the specification of each process. With

have thoroughly tested the implementation using different strategies prioritizing various types of actions (e.g., tending to process communication actions first).

However, the results showing the advantage of the random strategy and various anomaly results encountered during the testing phase inspire us to refine the extraction strategies, e.g making them more fine-grained. The extraction algorithm itself could be improved in the ways fruitful both for algorithm efficiency and the relevance of testing. It would be beneficial to implement a static analyzer for e.g. making predictions on a better extraction strategy based on a network structure. Also, more careful treatment of unextractable networks and their timely termination could help with the problem of noising tests with the data significantly altering from the average results.

Future is ephemeral

Continuing the topic of microservices coordination, we have investigated the problem of handling the data without storing it persistently, but ephemerally. Based on the formal model of MongoDB aggregation framework created by Botoeva et al. [41], we have developed a new model with the same set of operators, but with changes in syntax and semantics of data format and accompanying manipulations. We have implemented the model inside of the programming language Jolie for developing microservices-based applications.

One of the future work directions could be expanding TQuery by adding new syntax constructs for more expressiveness and ease of use of the language and by adding new operators to capture more complex queries (e.g. some aggregation pipeline operators like in MongoDB [2]). The evolution of the querying toolkit could also impact on enriching Jolie with new data formats.

We have already started the performance evaluation of our implementation, based on the eHealth scenario presented in [101]. We have compared the time-performance results of TQuery and MongoDB, as the nearest alternative in the field. One step here might be discovering the application and productivity of our solution for other contexts (IoT, electronic commerce, security), another is continuing comparing our performance results with other querying tools, both in-memory (e.g. LINQ), and DBMS-based (e.g. ArangoDB).

Bibliography

[1] Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[2] Aggregation pipeline operators in mongodb. https://docs.mongodb.com/manual/reference/operator/aggregation/.

[3] Amazon cognito. https://aws.amazon.com/cognito/.

[4] Apache kafka. distributed streaming platform. https://kafka.apache.org.

[5] Aws lambda. https://aws.amazon.com/lambda/.

[6] California Privacy and Protection law. https://www.caprivacy.org.

[7] Contract test. https://martinfowler.com/bliki/ContractTest.html.

[8] Cqrs. https://martinfowler.com/bliki/CQRS.html.

[9] Embracing the differences : Inside the netflix api redesign. https://netflixtechblog.com/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d.

[10] Flow: A static type checker for JavaScript. https://flowtype.org/.

[11] Ingenico. Fraud Detection Module Advanced: Checklist. "https://payment-services.ingenico.com/ogone/support/ /media/kdb/documents/fdmc_en.ashx?la=en".

[12] Iso/iec 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[13] Kong gateway. https://konghq.com/kong/.

[14] Kubernetes. production-grade container orchestration. https://kubernetes.io.

[15] Managing secrets in kubernetes. https://kubernetes.io/docs/concepts/configuration/secret/.

[16] Mesosphere dc/os. https://docs.d2iq.com.

[17] Nets. European Fraud Report – Payments Industry Challenges. https://www.nets.eu/solutions/fraud-and-dispute-services/Documents/Nets-Fraud-Report-2019.pdf.

[18] Pact. https://docs.pact.io.

[19] Russian Federal Law of 27 July 2006 N 152-FZ ON Personal Data. https://pd.rkn.gov.ru/authority/p146/p164/.

[20] Savara and testable architecture. https://savara.jboss.org.

[21] Security in mesosphere dc/os. https://docs.d2iq.com/mesosphere/dcos/1.12/security/.

[22] Stripe Radar. A primer on machine learning for fraud detection. https://stripe.com/en-fr/radar/guide.

[23] Vault. https://www.vaultproject.io.

[24] Evgenii Akentev, Alexander Tchitchigin, Larisa Safina, and Manuel Mazzara. Verified type-checker for jolie. https://arxiv.org/pdf/1703.05186.pdf.

[25] Edward B. Allen, Taghi M. Khoshgoftaar, and Yan Chen. Measuring coupling and cohesion of software modules: an information-theory approach. In Proceedings Seventh International Software Metrics Symposium, pages 124–134, 2001.

[26] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. 3:95–230, 07 2016.

[27] Apache Software Foundation. CouchDB Website. https://couchdb.apache.org/, 2018.

[28] ArangoDB. Arangodb. https://www.arangodb.com, 2014.

[29] Marco Autili, Paola Inverardi, and Massimo Tivoli. Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates. Science of Computer Programming, 160:3 – 29, 2018. Fundamentals of Software Engineering (selected papers of FSEN 2015).

[30] Stephanie B. Baker, Wei Xiang, and Ian Atkinson. Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities. IEEE Access, 5:26521–26544, 2017.

[31] Alexey Bandura, Nikita Kurilenko, Manuel Mazzara, Victor Rivera, Larisa Safina, and Alexander Tchitchigin. Jolie community on the rise. In 9th IEEE International Conference on Service-Oriented Computing and Applications, SOCA, 2016.

[32] Alistair Barros, Marlon Dumas, and Phillipa Oaks. A critical overview of the web services choreography description language. 09 2019.

[33] Massimo Bartoletti, Julien Lange, Alceste Scalas, and Roberto Zunino. Choreographies in the wild. Science of Computer Programming, 109:36 – 60, 2015. Selected Papers of the 6th Interaction and Concurrency Experience (ICE 2013).

[34] Len Bass. Software architecture in practice. Pearson Education India, 2007.

[35] Len Bass, Paulo Merson, and Liam O'Brien. Quality attributes and service-oriented architectures. Department of Defense, Technical Report September, 2005.

[36] S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In J. Field and M. Hicks, editors, POPL, pages 191–202. ACM, 2012.

[37] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In Proceedings of the 1995 Symposium on Software Reusability, SSR '95, pages 259–262, New York, NY, USA, 1995. ACM.

[38] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. SIGOPS Oper. Syst. Rev., 27(5):217–230, 1993.

[39] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In FMOODS/FORTE, 2013.

[40] Jan Bosch. Software architecture: The next step. In Software architecture, pages 194–199. Springer, 2004.

[41] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. Expressivity and complexity of mongodb queries. In ICDT, pages 9:1–9:23. Schloss Dagstuhl - LZI, 2018.

[42] Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/.

[43] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. J. ACM, 30(2):323–342, April 1983.

[44] Mario Bravetti and Gianluigi Zavattaro. Contract based multi-party service composition. In Proceedings of the 2007 International Conference on Fundamentals of Software Engineering, FSEN'07, page 207–222, Berlin, Heidelberg, 2007. Springer-Verlag.

[45] Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In Software Composition, pages 34–50. Springer Berlin Heidelberg, 2007.

[46] Brian Krebs. Extortionists Wipe Thousands of Databases, Victims Who Pay Up Get Stiffed. https://tinyurl.com/zt53ult, 2017.

[47] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. IEEE Trans. Softw. Eng., 25(1):91–121, January 1999.

[48] Frederick P Brooks. The mythical man-month, volume 1995. Addison-Wesley Reading, MA, 1975.

[49] Manfred Broy. A semantic and methodological essence of message sequence charts. Science of Computer Programming, 54(2-3):213–256, 2005.

[50] Tevfik Bultan and Xiang Fu. Choreography modeling and analysis with collaboration diagrams. IEEE Data Eng. Bull., 31:27–30, 01 2008.

[51] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In Proc. of COORDINATION, volume 4038 of LNCS, pages 63–81. Springer, 2006.

[52] Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, pages 74–95, 2016.

[53] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In CONCUR, volume 6269 of LNCS, pages 222–236. Springer, 2010.

[54] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst., 34(2):8, 2012.

[55] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée Desharnais and Radha Jagadeesan, editors, 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada, volume 59 of LIPIcs, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[56] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot, editors, POPL, pages 263–274. ACM, 2013.

[57] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. In P. Baldan and D. Gorla, editors, CONCUR, volume 8704 of LNCS, pages 47–62. Springer, 2014.

[58] Marco Carbone, Fabrizio Montesi, Carsten. Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. In L. Aceto and D. de Frutos-Escrig, editors, CONCUR, volume 42 of LIPIcs, pages 412–426. Schloss Dagstuhl, 2015.

[59] Marco Carbone, Fabrizio Montesi, and Hugo Vieira. Choreographies for reactive programming. 01 2018.

[60] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. LMCS, 8(1), 2012.

[61] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. TOCS, 26(2):4, 2008.

[62] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In Roberto Bruni and Vladimiro Sassone, editors, Trustworthy Global Computing, pages 25–45, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[63] James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. ACM SIGPLAN Notices, 48(9):403–416, 2013.

[64] Chor. Programming Language. http://www.chor-lang.org/.

[65] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.

[66] Toby Clemson. Testing strategies in a microservice architecture. https://martinfowler.com/articles/microservice-testing/.

[67] Jason Cohen, Eric Brown, Brandon DuRette, and Steven Teleki. Best kept secrets of peer code review. Smart Bear, 2006.

[68] B Terry Compton and Carol Withrow. Prediction and control of ada software defects. Journal of Systems and Software, 12(3):199–207, 1990.

[69] Melvin E Conway. How do committees invent. Datamation, 14(4):28–31, 1968.

[70] Apache CouchDB. Couchdb. https://couchdb.apache.org, 2005.

[71] Luís Cruz-Filipe and Fabrizio Montesi. Choreographies, computationally. CoRR, abs/1510.03271, 2015.

[72] Luís Cruz-Filipe and Fabrizio Montesi. Choreographies, divided and conquered. CoRR, abs/1602.03729, 2016.

[73] Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. The paths to choreography extraction. Foundations of Software Science and Computation Structures, pages 424–440, 2017.

[74] Andrea Dal Pozzolo, Olivier Caelen, Yann-Aël Le Borgne, Serge Waterschoot, and Gianluca Bontempi. Learned lessons in credit card fraud detection from a practitioner perspective. Expert Systems with Applications, 41:4915–4928, 08 2014.

[75] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. Logical Methods in Computer Science, 13, 11 2016.

[76] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. Logical Methods in Computer Science, 13, 11 2016.

[77] Florian Daniel. Web service orchestration and choreography: Enabling business processes on the web. E-business models, services, and communications, page 251, 01 2007.

[78] Harpal Dhama. Quantitative models of cohesion and coupling in software. J. Syst. Softw., 29(1):65–74, April 1995.

[79] Remco Dijkman and Marlon Dumas. Service-oriented design: A multi-viewpoint approach. International Journal of Cooperative Information Systems, 13, 12 2004.

[80] Edsger W. Dijkstra. On the Role of Scientific Thought, pages 60–66. Springer New York, New York, NY, 1982.

[81] Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen, and Manuel Mazzara. Microservices: Migration of a mission critical system. https://arxiv.org/abs/1704.04173.

[82] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In PAUSE, pages 195–216. Springer, 2017.

[83] Nicola Dragoni, Ivan Lanese, Stephan T. Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. Microservices: How to make your application scale. In A.P. Ershov Informatics Conference (the PSI Conference Series, 11th edition). Springer, 2017.

[84] Khaled El Emam, N Goel, W Melo, H Lounis, SN Rai, et al. The optimal class size for object-oriented software. Software Engineering, IEEE Transactions on, 28(5):494–509, 2002.

[85] Tom Ellis. Opaleye. https://github.com/tomjaguarpaw/haskell-opaleye, 2014.

[86] Michael Fagan. Design and code inspections to reduce errors in program development. In Software pioneers, pages 575–607. Springer, 2002.

[87] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine, 2000.

[88] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[89] Martin Fowler and Matthew Foemmel. Continuous integration, 2006. https://www.thoughtworks.com/continuous-integration.

[90] Martin Fowler and James Lewis. Microservices, 2014. http://martinfowler.com/articles/microservices.html.

[91] Mark Fussel. Foundations of object-relational mapping. ChiMu Corporation, 1997.

[92] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Self-reconfiguring microservices. In Theory and Practice of Formal Methods, pages 194–210. Springer, 2016.

[93] Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Guess Who's Coming: Runtime Inclusion of Participants in Choreographies, pages 118–138. 11 2019.

[94] Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Marco Peressotti, and Stefano Zingaro. No more, no less - a formal model for serverless computing, 03 2019.

[95] Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Stefano Pio Zingaro. A language-based approach for interoperability of iot platforms. In HICSS. AIS Electronic Library (AISeL), 2018.

[96] Maurizio Gabbrielli, Saverio Giallorenzo, and Fabrizio Montesi. Applied choreographies. CoRR, abs/1510.03637, 2015.

[97] Niall Gallagher. Cqengine. https://github.com/npgall/cqengine, 2009.

[98] Erich Gamma. Design patterns: elements of reusable object-oriented software. Pearson Education India, 1995.

[99] Jerry Gao. A reusable automated acceptance testing architecture for microservices in behavior-driven development. syntax, 1(6):7.

[100] Lars George. HBase: the definitive guide: random access to your planet-size data. " O'Reilly Media, Inc.", 2011.

[101] Saverio Giallorenzo, Fabrizio Montesi, Larisa Safina, and Stefano P. Zingaro. Ephemeral data handling in microservices. In 2019 IEEE International Conference on Services Computing (SCC), pages 234–236, July 2019.

[102] Jim Gray. A conversation with werner vogels. ACM Queue, 4(4):14–22, 2006.

[103] Jacob Grimm and Wilhelm Grimm. Snow white and the seven dwarfs.

[104] William Grosso. Java RMI. O'Reilly & Associates, Inc., 1st edition, 2001.

[105] Web Services Choreography Working Group et al. Web services choreography description language, 2002.

[106] Roberto Guanciale and Emilio Tuosto. An abstract semantics of the global view of choreographies. Electronic Proceedings in Theoretical Computer Science, 223:67–82, 08 2016.

[107] Claudio Guidi. Formalizing Languages for Service Oriented Computing. Ph.D. thesis, University of Bologna, 2007.

[108] Claudio Guidi, Ivan Lanese, Manuel Mazzara, and Fabrizio Montesi. Microservices: a language-based approach. In Present and Ulterior Software Engineering. Springer, 2017.

[109] Les Hatton. Reexamining the fault density-component size connection. IEEE software, 14(2):89–97, 1997.

[110] Pieter Hens, Monique Snoeck, Manu Backer, and Geert Poels. Decentralized event-based orchestration. volume 66, pages 695–706, 09 2010.

[111] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[112] Thomas T. Hildebrandt, Tijs Slaats, Hugo A. López, Søren Debois, and Marco Carbone. Declarative Choreographies and Liveness. In Jorge A. Pérez and Nobuko Yoshida, editors, 39th International Conference on Formal Techniques for Distributed Objects, Components, and System volume LNCS-11535 of Formal Techniques for Distributed Objects, Components, and Systems, pages 129–147, Copenhagen, Denmark, 2019. Springer International Publishing. Part 1: Full Papers.

[113] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. Citeseer, 1995.

[114] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In C. Hankin, editor, ESOP, volume 1381 of LNCS, pages 122–138. Springer, 1998.

[115] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. J. ACM, 63(1):9, 2016.

[116] Hans Hyttel, Emilio Tuosto, Hugo Vieira, Gianluigi Zavattaro, Ivan Lanese, Vasco Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, and António Ravara. Foundations of session types and behavioural contracts. ACM Computing Surveys, 49:1–36, 04 2016.

[117] Google Inc. Leveldb. http://leveldb.org, 2017.

[118] MongoDB Inc. Mongodb. https://www.mongodb.com, 2007.

[119] Joseph Ingeno. Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts. Packt Publishing, 2018.

[120] Michael Jang. Linux Annoyances for Geeks: Getting the Most Flexible System in the World Just the Way You Want It. " O'Reilly Media, Inc.", 2006.

[121] Jolie Developers Team. Jolie Website. https://www.jolie-lang.org/, 2018.

[122] J.P. Morgan. Payments fraud and control survey report. key highlights, 2018.

[123] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo. In Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, PPDP '16, page 146–159, New York, NY, USA, 2016. Association for Computing Machinery.

[124] Lucas Krause. Microservices: Patterns And Applications. Lucas Krause; 1 edition (April 1, 2015), 2014.

[125] Julien Lange and Emilio Tuosto. Synthesising choreographies from local session types. In CONCUR, pages 225–239, 2012.

[126] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In Sriram K. Rajamani and David Walker, editors, Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, pages 221–232. ACM, 2015.

[127] Vinh D. Le, Melanie M. Neff, Royal V. Stewart, Richard Kelley, Eric Fritzinger, Sergiu M. Dascalu, and Frederick C. Harris. Microservice-based architecture for the nrdc. In 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), pages 1659–1664, July 2015.

[128] Jing Li, Jifeng He, Geguang Pu, and Huibiao Zhu. Towards the semantics for web service choreography description language. In Proceedings of the 8th International Conference on Formal Methods and Software Engineering, ICFEM'06, pages 246–263, Berlin, Heidelberg, 2006. Springer-Verlag.

[129] Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, pages 195–211, 2016.

[130] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. J. Log. Algebr. Program., 70(1):96–118, 2007.

[131] Matthew C. MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, Rebekah Metz, and Booz Allen Hamilton. Reference model for service oriented architecture 1.0. OASIS Standard, 12, 2006.

[132] Tony Mauro. Adopting microservices at netflix: Lessons for team and process design. http://nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/, 2015.

[133] Manuel Mazzara. Towards Abstractions for Web Services Composition. Ph.D. thesis, University of Bologna, 2006.

[134] Manuel Mazzara and Sergio Govoni. A Case Study of Web Services Orchestration, pages 1–16. Springer Berlin Heidelberg, 2005.

[135] Dinesh P Mehta and Sartaj Sahni. Handbook of data structures and applications. Chapman and Hall/CRC, 2004.

[136] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In SIGMOD, pages 706–706. ACM, 2006.

[137] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239):2, 2014.

[138] Robin Milner. A Calculus of Communicating Systems, volume 92 of LNCS. Springer, 1980.

[139] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. Information and Computation, 100(1):1–40,41–77, September 1992.

[140] Bogdan Mingela, Nikolay Troshkov, Manuel Mazzara, Larisa Safina, and Alexander Tchitchigin. Towards static type-checking for jolie. https://arxiv.org/pdf/1702.07146.pdf.

[141] MongoDB Inc. MongoDB Aggregation Framework. https://docs.mongodb.com/manual/aggregation/, 2018.

[142] MongoDB Inc. MongoDB Website. https://www.mongodb.com/, 2018.

[143] Fabrizio Montesi. Choreographic Programming. Ph.D. thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

[144] Fabrizio Montesi. Process-aware web programming with jolie. Science of Computer Programming, 2016.

[145] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Jolie: a java orchestration language interpreter engine. Electronic Notes in Theoretical Computer Science, 181:19–33, 2007.

[146] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In A. Bouguettaya, Q.Z. Sheng, and F. Daniel, editors, Web Services Foundations, pages 81–107. Springer, 2014.

[147] Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In P.R. D'Argenio and H.C. Melgratti, editors, CONCUR, volume 8052 of LNCS, pages 425–439. Springer, 2013.

[148] Menno Mostert, Annelien L. Bredenoord, Monique C. I. H. Biesaart, and Johannes J. M. van Delden. Big data in medical research and eu data protection law: challenges to the consent or anonymise approach. European Journal Of Human Genetics, 24:956 EP, 2015.

[149] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. Microservice Architecture : Aligning Principles, Practices, and Culture. O'Reilly Media, Incorporated, 2016.

[150] Sam Newman. Building Microservices. "O'Reilly Media, Inc.", 2015.

[151] Sam Newman. What Are Microservices? "O'Reilly Media, Inc.", 2019.

[152] Niall Gallagher. CQEngine - Collection Query Engine. https://github.com/npgall/cqengine, 2018.

[153] Hyacinth S. Nwana. Software agents: an overview. The Knowledge Engineering Review, 11:205–244, 9 1996.

[154] OASIS. Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[155] OMG. Common Object Request Broker Architecture. http://www.omg.org/spec/CORBA/.

[156] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In 2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014., pages 305–319, 2014.

[157] Peter R Orszag. Evidence on the costs and benefits of health information technology. In testimony before Congress, volume 24, 2008.

[158] Anoop Panicker and Kishore Banala. Evolution of netflix conductor. https://netflixtechblog.com/evolution-of-netflix-conductor-16600be36bca.

[159] Chris Peltz. Web services orchestration and choreography. Computer, 36(10):46–52, Oct 2003.

[160] Dewayne E Perry and Alexander L Wolf. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes, 17(4):40–52, 1992.

[161] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt.

[162] Andrew Prunicki. Apache thrift, 2009.

[163] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In Proceedings of the 16th International Conference on World Wide Web, WWW '07, pages 973–982, New York, NY, USA, 2007. ACM.

[164] Eric S Raymond. The art of Unix programming. Addison-Wesley Professional, 2003.

[165] Chris Richardson. Microservices Patterns: With Examples in Java. Manning Publications, 2019.

[166] Roland Rieke, Maria Zhdanova, Jürgen Repp, Romain Giot, and Chrystel Gaber. Fraud detection in mobile payments utilizing process behavior analysis. In 2013 International Conference on Availability, Reliability and Security, pages 662–669, Sep. 2013.

[167] Richard Rodger. The Tao of Microservices. Manning Publications Company, 2017.

[168] Arnon Rotem-Gal-Oz. Fallacies of distributed computing explained. URL http://www.rgoarchitects. com/Files/fallacies. pdf, page 20, 2006.

[169] Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices (genericity in jolie). In Proc. of The 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2016.

[170] Salvatore Sanfilippo and Pieter Noordhuis. Redis. https://redis.io/, 2018.

[171] Scribble project home page. http://www.scribble.org.

[172] Mary Shaw and David Garlan. Software architecture: perspectives on an emerging discipline, volume 1. Prentice Hall Englewood Cliffs, 1996.

[173] Esther Shein. Ephemeral data. Comm. of the ACM, 56(9):20–22, 2013.

[174] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. IEEE Internet of Things Journal, 3(5):637–646, 2016.

[175] Yuqiong Sun, Susanta Nanda, and Trent Jaeger. Security-as-a-service for microservices-based cloud applications. In Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), CLOUDCOM '15, pages 50–57, Washington, DC, USA, 2015. IEEE Computer Society.

[176] Clemens Szyperski. Component Software: Beyond Object-oriented Programming. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[177] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[178] Alexander Tchitchigin, Larisa Safina, Manuel Mazzara, Mohamed Elwakil, Fabrizio Montesi, and Victor Rivera. Refinement types in jolie. In Spring/Summer Young Researchers Colloquium on Software Engineering, SYRCoSE, 2016.

[179] Omer Tene and Jules Polonetsky. Big data for all: Privacy and user control in the age of analytics. Nw. J. Tech. & Intell. Prop., 11:xxvii, 2012.

[180] Ed Tittel. The dangers of dark data and how to minimize your exposure. https://tinyurl.com/yb5lxc46, 2014.

[181] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, pages 350–369, 2013.

[182] Kenton Varda. Protocol buffers: Google's data interchange format. Google Open Source Blog, Available at least as early as Jul, 2008.

[183] Siva Visveswaran. Dive into connection pooling with j2ee. reprinted from JavaWorld, 7, 2000.

[184] W3C. Web services architecture. http://www.w3.org/TR/ws-arch/.

[185] W3C. Web Services Choreography Description Language. https://www.w3.org/TR/ws-cdl-10/.

[186] W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.

[187] Philip Wadler. Propositions as sessions. 24(2–3):384–418, 2014. Also: ICFP, pages 273–286, 2012.

[188] Allen Wang and Sudhir Tonse. Announcing ribbon: Tying the netflix mid-tier services together, January 2013. http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html.

[189] Charles Webster. From APIs to Microservices: Workflow Orchestration and Choreography Across Healthcare Organizations. https://tinyurl.com/ya57w7wu, 2016.

[190] Luke Welling and Laura Thomson. PHP and MySQL Web development. Sams Publishing, 2003.

[191] Oliver Wolf. Introduction into microservices. https://specify.io/concepts/microservices, 2019.

[192] Zhixian Yan, Manuel Mazzara, Emilia Cimpian, and Alexander Urbanec. Business process modeling: Classifications and perspectives. In Business Process and Services Computing: 1st International Working Conference on Business Process and Services Computing, BPSC 2007, September 25-26, 2007, Leipzig, Germany., page 222, 2007.

[193] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers, pages 22–41, 2013.

Microservices: yesterday, today, and tomorrow

Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara
Fabrizio Montesi, Ruslan Mustafin, Larisa Safina

Abstract Microservices is an architectural style inspired by service-oriented computing that has recently started gaining popularity. Before presenting the current state-of-the-art in the field, this chapter reviews the history of software architecture, the reasons that led to the diffusion of objects and services first, and microservices later. Finally, open problems and future challenges are introduced. This survey primarily addresses newcomers to the discipline, while offering an academic viewpoint on the topic. In addition, we investigate some practical issues and point out some potential solutions.

1 Introduction

The mainstream languages for development of server-side applications, like Java, C/C++, and Python, provide abstractions to break down the complexity of programs into modules. However, these languages are designed for the creation of *single executable artefacts*, also called *monoliths*, and their modularisation abstractions rely on the sharing of resources of the same machine (memory, databases, files). Since the modules of a monolith depend on said shared resources, they are not independently executable.

Definition 1 (Monolith). A monolith is a software application whose modules cannot be executed independently.

This makes monoliths difficult to use in distributed systems without specific frameworks or ad hoc solutions such as, for example, Network Objects [8], RMI [41] or CORBA [69]. However, even these approaches still suffer from the general issues that affect monoliths; below we list the most relevant ones (we label issues **I**$\lfloor n \rfloor$):

Nicola Dragoni, Alberto Lluch Lafuente
Technical University of Denmark e-mail: {ndra,albl}@dtu.dk

Saverio Giallorenzo
INRIA/Univ. of Bologna, Dept. of Computer Science and Engineering, Italy e-mail: saverio.giallorenzo@gmail.com

Manuel Mazzara, Ruslan Mustafin, Larisa Safina
Innopolis University, Russian Federation e-mail: {m.mazzara,r.mustafin,l.safina}@innopolis.ru

Fabrizio Montesi
University of Southern Denmark e-mail: fmontesi@imada.sdu.dk

I|1 large-size monoliths are difficult to maintain and evolve due to their complexity. Tracking down bugs requires long perusals through their code base;
I|2 monoliths also suffer from the "dependency hell" [57], in which adding or updating libraries results in inconsistent systems that do not compile/run or, worse, misbehave;
I|3 any change in one module of a monolith requires rebooting the whole application. For large-sized projects, restarting usually entails considerable downtimes, hindering development, testing, and the maintenance of the project;
I|4 deployment of monolithic applications is usually sub-optimal due to conflicting requirements on the constituent models' resources: some can be memory-intensive, others computational-intensive, and others require ad-hoc components (e.g., SQL-based rather than graph-based databases). When choosing a deployment environment, the developer must compromise with a one-size-fits-all configuration, which is either expensive or sub-optimal with respect to the individual modules;
I|5 monoliths limit scalability. The usual strategy for handling increments of inbound requests is to create new instances of the same application and to split the load among said instances. However, it could be the case that the increased traffic stresses only a subset of the modules, making the allocation of the new resources for the other components inconvenient;
I|6 monoliths also represent a technology lock-in for developers, which are bound to use the same language and frameworks of the original application.

The *microservices* architectural style [35] has been proposed to cope with such problems. In our definition of microservice, we use the term "cohesive" [27, 46, 7, 11, 3] to indicate that a service implements only functionalities strongly related to the concern that it is meant to model.

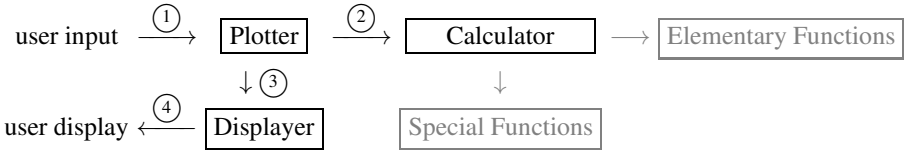
Definition 2 (Microservice). A microservice is a cohesive, independent process interacting via messages.

As an example, consider a service intended to compute calculations. To call it a microservice, it should provide arithmetic operations requestable via messages, but it should not provide other (possibly loosely related) functionalities like plotting and displaying of functions.

From a technical point of view, microservices should be independent components conceptually deployed in isolation and equipped with dedicated memory persistence tools (e.g., databases). Since all the components of a microservice architecture are microservices, its distinguishing behaviour derives from the composition and coordination of its components via messages.

Definition 3 (Microservice Architecture). A microservice architecture is a distributed application where all its modules are microservices.

To give an example of a microservice architecture, let us assume that we want to provide a functionality that plots the graph of a function. We also assume the presence of two microservices: Calculator and Displayer. The first is the calculator microservice mentioned above, the second renders and displays images. To fulfil our goal, we can introduce a new microservice, called Plotter, that orchestrates Calculator to calculate the shape of the graph and that invokes Displayer to render the calculated shape. Below, we report (in black) a depiction of the workflow of such a microservice architecture.



The developers of the architecture above can focus separately on implementing the basic microservice functionalities, i.e., the Calculator and the Displayer. Finally, they can implement the behaviour of the distributed application with the Plotter that ① takes the function given by a user, ② interacts with the Calculator to compute a symbolic representation of the graph of the function, and finally ③ requests the Displayer to show the result back to the user ④. To illustrate how the microservice approach scales by building on pre-existing microservice architectures, in the figure above we drew the Calculator orchestrating two extra microservices (in grey) that implement mathematical Elementary and Special Functions.

The microservice architectural style does not favour or forbid any particular programming paradigm. It provides a guideline to partition the components of a distributed application into independent entities, each addressing one of its concerns. This means that a microservice, provided it offers its functionalities via message passing, can be internally implemented with any of the mainstream languages cited in the beginning of this section.

The principle of microservice architectures assists project managers and developers: it provides a guideline for the design and implementation of distributed applications. Following this principle, developers focus on the implementation and testing of a few, cohesive functionalities. This holds also for higher-level microservices, which are concerned with coordinating the functionalities of other microservices.

We conclude this section with an overview, detailed in greater depth in the remainder of the paper, on how microservices cope with the mentioned issues of monolithic applications (below, **S**|n is a solution to issue **I**|n).

S|1 microservices implement a limited amount of functionalities, which makes their code base small and inherently limits the scope of a bug. Moreover, since microservices are independent, a developer can directly test and investigate their functionalities in isolation with respect to the rest of the system;

S|2 it is possible to plan *gradual transitions* to new versions of a microservice. The new version can be deployed "next" to the old one and the services that depend on the latter can be gradually modified to interact with the former. This fosters *continuous integration* [34] and greatly eases software maintenance;

S|3 as a consequence of the previous item, changing a module of a microservice architecture does not require a complete reboot of the whole system. The reboot regards only the microservices of that module. Since microservices are small in size, programmers can develop, test, and maintain services experiencing only very short re-deployment downtimes;

S|4 microservices naturally lend themselves to containerisation [58], and developers enjoy a high degree of freedom in the configuration of the deployment environment that best suits their needs (both in terms of costs and quality of service);

S|5 scaling a microservice architecture does not imply a duplication of all its components and developers can conveniently deploy/dispose instances of services with respect to their load [36];

S|6 the only constraint imposed on a network of interoperating microservices is the technology used to make them communicate (media, protocols, data encodings). Apart from that, microservices impose no additional lock-in and developers can freely choose the optimal resources (languages, frameworks, etc.) for the implementation of each microservice.

In the remainder of this paper, in § 2, we give a brief account of the evolution of distributed architectures until their recent incarnation in the microservice paradigm. Then, we detail the problems that microservices can solve and their proposed solutions in the form of microservice architectures. In § 3, we detail the current solutions for developing microservice architectures and how microservices affect the process of software design, development, testing, and maintenance. In § 4 we discuss the open challenges and the desirable tools for programming microservice architecture. In § 5 we draw overall conclusions.

2 Yesterday

Architecture is what allows systems to evolve and provide a certain level of service throughout their life-cycle. In software engineering, architecture is concerned with providing a bridge between system functionality and requirements for quality attributes that the system has to meet. Over the past several decades, software architecture has been thoroughly studied, and as a result software engineers have come up with different ways to compose systems that provide broad functionality and satisfy a wide range of requirements. In this section, we provide an overview of the work on software architectures from the early days to the advent of microservices.

2.1 From the early days to Object-oriented design patterns

The problems associated with large-scale software development were first experienced around the 1960s [12]. The 1970s saw a huge rise of interest from the research community for software design and its implications on the development process. At the time, the design was often considered as an activity not associated with the implementation itself and therefore requiring a special set of notations and tools. Around the 1980s, the full integration of design into the development processes contributed towards a partial merge of these two activities, thus making it harder to make neat distinctions.

References to the concept of software architecture also started to appear around the 1980s. However, a solid foundation on the topic was only established in 1992 by Perry and Wolf [72]. Their definition of software architecture was distinct from software design, and since then it has generated a large community of researchers studying the notion and the practical applications of software architecture, allowing the concepts to be widely adopted by both industry and academia.

This spike of interest contributed to an increase in the number of existing software architecture patterns (or generally called *styles*), so that some form of classification was then required. This problem was tackled in one of the most notable works in the field, the book "Software Architecture: Perspectives on an Emerging Discipline" by Garlan and Shaw [77]. Bosch's work [9] provides a good overview of the current research state in software engineering and architecture. Since its appearance in the 1980s, software architecture has developed into a mature discipline making use of notations, tools, and several techniques. From the pure, and occasionally speculative, realm of academic basic research, it has made the transition into an element that is essential to industrial software construction.

The advent and diffusion of object-orientation, starting from the 1980s and in particular in the 1990s, brought its own contribution to the field of Software Architecture. The classic by Gamma et al. [38] covers the design of object-oriented software and how to translate it into code presenting a collection of recurring solutions, called *patterns*. This idea is neither new nor exclusive to Software Engineering, but the book is the first compendium to popularize the idea on a large scale. In the pre-Gamma era patterns for OO solutions were already used: a typical example of an architectural design pattern in object-oriented programming is the Model-View-Controller (MVC) [33], which has been one of the seminal insights in the early development of graphical user interfaces.

2.2 Service-oriented Computing

Attention to *separation of concerns* has recently led to the emergence of the so-called Component-based software engineering (CBSE) [79], which has given better control over design, implementation and evolution of software systems. The last decade has seen a further shift towards the concept of service first [83] and the natural evolution to microservices afterwards.

Service-Oriented Computing (SOC) is an emerging paradigm for distributed computing and e-business processing that finds its origin in object-oriented and component computing. It has been introduced to harness the complexity of distributed systems and to integrate different software applications [53]. In SOC, a program — called a *service* — offers functionalities to other components, accessible via message passing. Services decouple their interfaces (i.e. how other services access their functionalities) from their implementation. On top of that, specific workflow languages are then defined in order to orchestrate the complex actions of services (e.g. WS-BPEL [68]). These languages share ideas with some well-known formalisms from concurrency theory, such as CCS and the π-calculus [59, 60]. This aspect fostered the development of formal models for better understanding and verifying service interactions, ranging from foundational process models of SOC [55, 43, 52] to theories for the correct composition of services [10, 47, 48]. In [87] a classification of approaches for business modeling puts this research in perspective.

The benefits of service-orientation are:

- **Dynamism** - New instances of the same service can be launched to split the load on the system;
- **Modularity and reuse** - Complex services are composed of simpler ones. The same services can be used by different systems;
- **Distributed development** - By agreeing on the interfaces of the distributed system, distinct development teams can develop partitions of it in parallel;
- **Integration of heterogeneous and legacy systems** - Services merely have to implement standard protocols to communicate.

2.3 Second generation of services

The idea of componentization used in service-orientation can be partially traced back to the object-oriented programming (OOP) literature; however, there are peculiar differences that led to virtually separate research paths and communities. As a matter of fact, SOC at the origin was - and still is - built on top of OOP languages, largely due to their broad diffusion in the early 2000s. However, the evolution of objects into services, and the relative comparisons, has to be treated carefully since the first focus on encapsulation and information is hidden in a *shared-memory* scenario, while the second is built on the idea of independent deployment and *message-passing*. It is therefore a paradigm shift, where both the paradigms share the common idea of componentization. The next step is adding the notion of *business capability* and therefore focusing analysis and design on it so that the overall system architecture is determined on this basis.

The first "generation" of service-oriented architectures (SOA) defined daunting and nebulous requirements for services (e.g., discoverability and service contracts), and this hindered the adoption of the SOA model. Microservices are the second iteration on the concept of SOA and SOC. The aim is to strip away unnecessary levels of complexity in order to focus on the programming of simple services that effectively implement a single functionality. Like OO, the microservices paradigm needs ad-hoc tools to support devel-

opers and naturally leads to the emergence of specific design patterns [76]. First and foremost, languages that embrace the service-oriented paradigm are needed (instead, for the most part, microservice architectures still use OO languages like Java and Javascript or functional ones). The same holds for the other tools for development support like testing suites, (API) design tools, etc.

3 Today

The microservices architecture appeared lately as a new paradigm for programming applications by means of the composition of small services, each running its own processes and communicating via light-weight mechanisms. This approach has been built on the concepts of SOA [53] brought from crossing-boundaries workflows to the application level and into the applications architectures, i.e. its Service-Oriented Architecture and Programming from the large to the small.

The term "microservices" was first introduced in 2011 at an architectural workshop as a way to describe the participants' common ideas in software architecture patterns [35]. Until then, this approach had also been known under different names. For example, Netflix used a very similar architecture under the name of *Fine grained SOA* [86].

Microservices now are a new trend in software architecture, which emphasises the design and development of highly maintainable and scalable software. Microservices manage growing complexity by functionally decomposing large systems into a set of independent services. By making services completely independent in development and deployment, microservices emphasise loose coupling and high cohesion by taking modularity to the next level. This approach delivers all sorts of benefits in terms of maintainability, scalability and so on. It also comes with a bundle of problems that are inherited from distributed systems and from SOA, its predecessor. The Microservices architecture still shows distinctive characteristics that blend into something unique and different from SOA itself:

- **Size** - The size is comparatively small wrt. a typical service, supporting the belief that the architectural design of a system is highly dependent on the structural design of the organization producing it. Idiomatic use of the microservices architecture suggests that if a service is too large, it should be split into two or more services, thus preserving granularity and maintaining focus on providing only a single business capability. This brings benefits in terms of service maintainability and extendability.
- **Bounded context** - Related functionalities are combined into a single business capability, which is then implemented as a service.
- **Independency** - Each service in microservice architecture is operationally independent from other services and the only form of communication between services is through their published interfaces.

The key system characteristics for microservices are:

- **Flexibility** - A system is able to keep up with the ever-changing business environment and is able to support all modifications that is necessary for an organisation to stay competitive on the market
- **Modularity** - A system is composed of isolated components where each component contributes to the overall system behaviour rather than having a single component that offers full functionality
- **Evolution** - A system should stay maintainable while constantly evolving and adding new features

The microservices architecture gained popularity relatively recently and can be considered to be in its infancy since there is still a lack of consensus on what microservices actually are. M. Fowler and J. Lewis provide a starting ground by defining principal characteristics of microservices [35]. S. Newman [66] builds upon M. Fowler's article and presents recipes and best practices regarding some aspects of the aforementioned architecture. L. Krause in his work [49] discusses patterns and applications of microservices. A number of papers has also been published that describe details of design and implementation of systems using microservices architecture. For example, the authors of [50] present development details of a new software system for Nevada Research Data Center (NRDC) using the microservices architecture. M. Rahman and J. Gao in [39] describe an application of behaviour-driven development (BDD) to the microservices architecture in order to decrease the maintenance burden on developers and encourage the usage of acceptance testing.

3.1 Teams

Back in 1968, Melvin Conway proposed that an organisation's structure, or more specifically, its communication structure constrains a system's design such that the resulting design is a copy of the organisation's communication patterns [24]. The microservices approach is to organise cross-functional teams around services, which in turn are organised around business capabilities [35]. This approach is also known as "you build, you run it" principle, first introduced by Amazon CTO Werner Vogels [40]. According to this approach, teams are responsible for full support and development of a service throughout its lifecycle.

3.2 Total automation

Each microservice may represent a single business capability that is delivered and updated independently and on its own schedule. Discovering a bug and or adding a minor improvement do not have any impact on other services and on their release schedule (of course, as long as backwards compatibility is preserved and a service interface remains unchanged). However, to truly harness the power of independent deployment, one must utilise very efficient integration and delivery mechanisms. This being said, microservices are the first architecture developed in the post-continuous delivery era and essentially microservices are meant to be used with continuous delivery and continuous integration, making each stage of delivery pipeline automatic. By using automated continuous delivery pipelines and modern container tools, it is possible to deploy an updated version of a service to production in a matter of seconds [54], which proves to be very beneficial in rapidly changing business environments.

3.3 Choreography over orchestration

As discussed earlier, microservices may cooperate in order to provide more complex and elaborate functionalities. There are two approaches to establish this cooperation – orchestration [56] and choreography [71]. Orchestration requires a conductor – a central service that will send requests to other services and oversee the process by receiving responses. Choreography, on the other hand, assumes no centralisation and uses events

and publish/subscribe mechanisms in order to establish collaboration. These two concepts are not new to microservices, but rather are inherited from the SOA world where languages such as WS-BPEL [68] and WS-CDL [84] have long represented the major references for orchestration and choreography respectively (with vivid discussions between the two communities of supporters).

Prior to the advent of microservices and at the beginning of the SOA's hype in particular, orchestration was generally more popular and widely adopted, due to its simplicity of use and easier ways to manage complexity. However, it clearly leads to service coupling and uneven distribution of responsibilities, and therefore some services have a more centralising role than others. Microservices' culture of decentralisation and the high degrees of independence represents instead the natural application scenario for the use of choreography as a means of achieving collaboration. This approach has indeed recently seen a renewed interest in connection with the broader diffusion of microservices in what can be called the *second wave of services*.

3.4 Impact on quality and management

In order to better grasp microservices we need to understand the impact that this architecture has on some software quality attributes.

Availability

Availability is a major concern in microservices as it directly affects the success of a system. Given services independence, the whole system availability can be estimated in terms of the availability of the individual services that compose the system. Even if a single service is not available to satisfy a request, the whole system may be compromised and experience direct consequences. If we take service implementation, the more fault-prone a component is, the more frequently the system will experience failures. One would argue that small-size components lead to a lower fault density. However, it has been found by Hatton [44] and by Compton and Withrow [23] that small-size software components often have a very high fault density. On the other hand, El Emam *et al.* in their work [30] found that as size increases, so does a component's fault proneness. Microservices are prevented from becoming too large as idiomatic use of the microservices architecture suggests that, as a system grows larger, microservices should be prevented from becoming overly complex by refining them into two or more different services. Thus, it is possible to keep optimal size for services, which may theoretically increase availability. On the other hand, spawning an increasing number of services will make the system fault-prone on the integration level, which will result in decreased availability due to the large complexity associated with making dozens of services instantly available.

Reliability

Given the distributed nature of the microservices architecture, particular attention should be paid to the reliability of message-passing mechanisms between services and to the reliability of the services themselves. Building the system out of small and simple components is also one of the rules introduced in [74], which states that in order to achieve higher reliability one must find a way to manage the complexities of a large system: building things out of simple components with clean interfaces is one way to achieve this. The

greatest threat to microservices reliability lies in the domain of integration and therefore when talking about microservices reliability, one should also mention integration mechanisms. One example of this assumption being false is using a network as an integration mechanism and assuming network reliability is one of the first fallacies of distributed computing [75]. Therefore, in this aspect, microservices reliability is inferior to the applications that use in-memory calls. It should be noted that this downside is not unique only to microservices and can be found in any distributed system. When talking about messaging reliability, it is also useful to remember that microservices put restrictions on integration mechanisms. More specifically, microservices use integration mechanisms in a very straightforward way - by removing all functionality that is not related to the message delivering and focusing solely on reliable message delivery.

Maintainability

By nature, the microservices architecture is loosely coupled, meaning that there is a small number of links between services and services themselves being independent. This greatly contributes to the maintainability of a system by minimising the costs of modifying services, fixing errors or adding new functionality. Despite all efforts to make a system as maintainable as possible, it is always possible to spoil maintainability by writing obscure and counterintuitive code [5]. As such, another aspect of microservices that can lead to increased maintainability is the above mentioned "you build it, you run it" principle, which leads to better understanding a given service, its business capabilities and roles [31, 22].

Performance

The prominent factor that negatively impacts performance in the microservices architecture is communication over a network. The network latency is much greater than that of memory. This means that in-memory calls are much faster to complete than sending messages over the network. Therefore, in terms of communication, the performance will degrade compared to applications that use in-memory call mechanisms. Restrictions that microservices put on size also indirectly contribute to this factor. In more general architectures without size-related restrictions, the ratio of in-memory calls to the total number of calls is higher than in the microservices architecture, which results in less communication over the network. Thus, the exact amount of performance degradation will also depend on the system's interconnectedness. As such, systems with well-bounded contexts will experience less degradation due to looser coupling and fewer messages sent.

Security

In any distributed system security becomes a major concern. In this sense, microservices suffer from the same security vulnerabilities as SOA [6]. As microservices use REST mechanism and XML with JSON as main data-interchange formats, particular attention should be paid to providing security of the data being transferred. This means adding additional overhead to the system in terms of additional encryption functionality. Microservices promote service reuse, and as such it is natural to assume that some systems will include third-party services. Therefore, an additional challenge is to provide authentication mechanisms with third-party services and ensure that the sent data is stored securely. In summary, microservices' secu-

rity is impacted in a rather negative manner because one has to consider and implement additional security mechanisms to provide additional security functionality mentioned above.

Testability

Since all components in a microservices architecture are independent, each component can be tested in isolation, which significantly improves component testability compared to monolithic architecture. It also allows to adjust the scope of testing based on the size of changes. This means that with microservices it is possible to isolate parts of the system that changed and parts that were affected by the change and to test them independently from the rest of the system. Integration testing, on the other hand, can become very tricky, especially when the system that is being tested is very large, and there are too many connections between components. It is possible to test each service individually, but anomalies can emerge from collaboration of a number of services.

4 Tomorrow

Microservices are so recent that we can consider their exploration to have just begun. In this section, we discuss interesting future directions that we envision will play key roles in the advancement of the paradigm.

The greatest strength of microservices comes from pervasive distribution: even the internal components of software are autonomous services, leading to loosely coupled systems and the other benefits previously discussed. However, from this same aspect (distribution) also comes its greatest weakness: programming distributed systems is inherently harder than monoliths. We now have to think about new issues. Some examples are: how can we manage changes to a service that may have side-effects on the other services that it communicates with? How can we prevent attacks that exploit network communications?

4.1 Dependability

There are many pitfalls that we need to keep in mind when programming with microservices. In particular, preventing programming errors is hard. Consequently, building dependable systems is challenging.

Interfaces

Since microservices are autonomous, we are free to use the most appropriate technology for the development of each microservice. A disadvantage introduced by this practice is that different technologies typically have different means of specifying contracts for the composition of services (e.g., interfaces in Java, or WSDL documents in Web Services [20]). Some technologies do not even come with a specification language and/or a compatibility checker of microservices (Node.js, based on JavaScript, is a prime example).

Thus, where do we stand? Unfortunately, the current answer is informal documentation. Most services come with informal documents expressed in natural language that describe how clients should use the service. This makes the activity of writing a client very error-prone, due to potential ambiguities. Moreover,

we have no development support tools to check whether service implementations actually implement their interfaces correctly.

As an attempt to fix this problem, there are tools for the formal specification of message types for data exchange, which one can use to define service interfaces independently of specific technologies. Then, these technology-agnostic specifications can be either compiled to language-specific interfaces — e.g., compiling an interface to a Java type — or used to check for well-typedness of messages (wrt. interfaces and independently of the transport protocol). Examples of tools offering these methodologies are Jolie [64, 4, 21], Apache Thrift [73], and Google's Protocol Buffers [82]. However, it is still unclear how to adapt tools to implement the mechanical checking (at compile or execution time) of messages for some widespread architectural styles for microservices, such as REST [32], where interfaces are constrained to a fixed set of operations and actions are expressed on dynamic resource paths. A first attempt at bridging the world of technology-agnostic interfaces based on operations and REST is presented in [63], but checking for the correctness of the binding information between the two is still left as a manual task to the programmer. Another, and similar, problem is trying to apply static type checking to dynamic languages (e.g., JavaScript and Jolie), which are largely employed in the development of microservices [1, 61, 2].

Behavioural Specifications and Choreographies

Having formally-defined interfaces in the form of an API is not enough to guarantee the compatibility of services. This is because, during execution, services may engage in sessions during which they perform message exchanges in a precise order. If two services engage in a session and start performing incompatible I/O, this can lead to various problems. Examples include: a client sending a message on a stream that was previously closed; deadlocks, when two services expect a message from one another without sending anything; or, a client trying to access an operation that is offered by a server only after a successful distributed authentication protocol with a third-party is performed.

Behavioural types are types that can describe the behaviour of services and can be used to check that two (or more) services have compatible actions. Session types are a prime example of behavioural types [47, 48]. Session types have been successfully applied to many contexts already, ranging from parallel to distributed computing. However, no behavioural type theory is widely adopted in practice yet. This is mainly because behavioural types restrict the kind of behaviours that programmers can write for services, limiting their applicability. An important example of a feature with space for improvement is non-determinism. In many interesting protocols, like those for distributed agreement, execution is non-deterministic and depending on what happens at runtime, the participants have to react differently [70].

Behavioural interfaces are a hot topic right now and will likely play an important role in the future of microservices. We envision that they will also be useful for the development of automatic testing frameworks that check the communication behaviour of services.

Choreographies

Choreographies are high-level descriptions of the communications that we want to happen in a system in contrast with the typical methodology of defining the behaviour of each service separately. Choreographies are used in some models for behavioural interfaces, but they actually originate from efforts at the W3C of defining a language that describes the global behaviour of service systems [42]. Over the past decade, choreographies have been investigated for supporting a new programming paradigm called Choreographic

Programming [62]. In Choreographic Programming, the programmer uses choreographies to program service systems and then a compiler is used to automatically generate compliant implementations. This yields a correctness-by-construction methodology, guaranteeing important properties such as deadlock-freedom and lack of communication errors [15, 17, 65].

Choreographies may have an important role in the future of microservices, since they shrink the gap between requirements and implementations, making the programmer able to formalise the communications envisioned in the design phase of software. Since the correctness of the compiler from choreographies to distributed implementations is vital in this methodology, formal models are being heavily adopted to develop correct compilation algorithms [37]. However, a formalisation of how transparent mobility of processes from one protocol to the other is still missing. Moreover, it is still unclear how choreographies can be combined with flexible deployment models where nodes may be replicated or fail at runtime. An initial investigation on the latter is given in [51]. Also, choreographies are still somewhat limited in expressing non-deterministic behaviour, just like behavioural types.

Moving Fast with Solid Foundations

Behavioural types, choreographies, refinement types [80] and other models address the problem of specifying, verifying, and synthesising communication behaviours. However, there is still much to be discovered and developed on these topics. It is then natural to ask: do we really need to start these investigations from scratch? Or, can we hope to reuse results and structures from other well-established models in Computer Science?

A recent line of work suggests that a positive answer can be found by connecting behavioural types and choreographies to well-known logical models. A prominent example is a Curry-Howard correspondence between session types and the process model of π-calculus, given in [14] (linear logical propositions correspond to session types, and communications to proof normalization in linear logic). This result has propelled many other results, among which: a logical reconstruction of behavioural types in classical linear logic that supports parametric polymorphism [85]; type theories for integrating higher-order process models with functional computation [81]; initial ideas for algorithms for extracting choreographies from separate service programs [18]; a logical characterisation of choreography-based behavioural types [19]; and, explanations of how interactions among multiple services (multiparty sessions) are related to well-known techniques for logical reasoning [16, 13].

Another principle that we can use for the evolution of choreographic models is the established notion of computation. The minimal set of language features to achieve Turing completeness in choreographies is known [25]. More relevant in practice, this model was used to develop a methodology of procedural programming for choreographies, allowing for the writing of correct-by-construction implementations of divide-and-conquer distributed algorithms [26].

We can then conclude that formal methods based on well-known techniques seem to be a promising starting point for tackling the issue of writing correct microservice systems. This starting point gives us solid footing for exploring the more focused disciplines that we will need in the future, addressing problems like the description of coordination patterns among services. We envision that these patterns will benefit from the rich set of features that formal languages and process models have to offer, such as expressive type theories and logics. It is still unclear, however, how exactly these disciplines can be extended to naturally capture the practical scenarios that we encounter in microservices. We believe that empirically investigating microservice programming will be beneficial in finding precise research directions in this regard.

4.2 Trust and Security

The microservices paradigm poses a number of trust and security challenges. These issues are certainly not new, as they apply to SOA and in general to distributed computing, but they become even more challenging in the context of microservices. In this section, we aim to discuss some of these key security issues.

Greater Surface Attack Area

In monolithic architectures, application processes communicate via internal data structures or internal communication (for instance, socket or RMI). The attack surface is usually also constrained to a single OS. On the contrary, the microservices paradigm is characterised by applications that are broken down into services that interact with each other through APIs exposed to the network. APIs are independent of machine architectures and even programming languages. As a result, they are exposed to more potential attacks than traditional subroutines or functionalities of a large application, which only interacted with other parts of the same application. Moreover, application internals (the microservices) have now become accessible from the external world. Rephrasing, this means that microservices can in principle send the attack surface of a given application through the roof.

Network Complexity

The microservices vision, based on the creation of many small independent applications interacting with each other, can result in complex network activity. This network complexity can significantly increase the difficulty in enforcing the security of the overall microservices-based application. Indeed, when a real-world application is decomposed, it can easily create hundreds of microservices, as seen in the architecture overview of Hailo, an online cab reservation application.[1] Such an intrinsic complexity determines an ever-increasing difficulty in debugging, monitoring, auditing, and forensic analysis of the entire application. Attackers could exploit this complexity to launch attacks against applications.

Trust

Microservices, at least in this early stage of development, are often designed to completely trust each other. Considering a microservice trustworthy represents an extremely strong assumption in the "connectivity era", where microservices can interact with each other in a heterogeneous and open way. An individual microservice may be attacked and controlled by a malicious adversary, compromising not only the single microservice but, more drastically, bringing down the entire application. As an illustrative real world example, a subdomain of Netflix was recently compromised, and from that domain, an adversary can serve any content in the context of `netflix.com`. In addition, since Netflix allowed all users' cookies to be accessed from any subdomain, a malicious individual controlling a subdomain was able to tamper with authenticated Netflix subscribers and their data [78]. Future microservices platforms need mechanisms to monitor and enforce the connections among microservices to confine the trust placed on individual microservices, limiting the potential damage if any microservice gets compromised.

[1] `hailoapp.com`

Heterogeneity

The microservices paradigm brings heterogeneity (of distributed systems) to its maximum expression. Indeed, a microservices-based system can be characterised by: a large number of autonomous entities that are not necessarily known in advance (again, trust issue); a large number of different administrative security domains, creating competition amongst providers of different services; a large number of interactions across different domains (through APIs); no common security infrastructure (different "Trusted Computing Base"); and last but not least, no global system to enforce rules.

The research community is still far from adequately addressing the aforementioned security issues. Some recent works, like [78], show that some preliminary contribution is taking place. However, the challenge of building secure and trustworthy microservices-based systems is still more than open.

5 Conclusions

The microservice architecture is a style that has been increasingly gaining popularity in the last few years, both in academia and in the industrial world. In particular, the shift towards microservices is a sensitive matter for a number of companies involved in a major refactoring of their back-end systems [29].

Despite the fact that some authors present it from a *revolutionary* perspective, we have preferred to provide an *evolutionary* presentation to help the reader understand the main motivations that lead to the distinguishing characteristics of microservices and relate to well-established paradigms such as OO and SOA. With microservice architecture being very recent, we have not found a sufficiently comprehensive collection of literature in the field, so that we felt the need to provide a starting point for newcomers to the discipline, and offer the authors' viewpoint on the topic.

In this chapter, we have presented a (necessarily incomplete) overview of software architecture, mostly providing the reader with references to the literature, and guiding him/her in our itinerary towards the advent of services and microservices. A specific arc has been given to the narrative, which necessarily emphasises some connections and some literature, and it is possibly too severe with other sources. For example, research contributions in the domain of the actor model [45] and software agents [67] have not been emphasised enough, and still modern distributed systems have been influenced by these communities too. This calls for a broader survey investigating relationships along this line. For information on the relation between microservices and scalability, the reader may refer to [28].

Acknowledgements

Montesi was supported by CRC (Choreographies for Reliable and efficient Communication software), grant no. DFF–4005-00304 from the Danish Council for Independent Research. Giallorenzo was supported by the EU EIT Digital project SMAll. This work has been partially funded by an Erasmus Mundus Scholarship. We would like to thank Daniel Martin Johnston who played a major role in proofreading the final draft of the paper and improving the quality of writing.

References

1. Flow: A static type checker for JavaScript. https://flowtype.org/.
2. Evgenii Akentev, Alexander Tchitchigin, Larisa Safina, and Manuel Mazzara. Verified type-checker for jolie. `https://arxiv.org/pdf/1703.05186.pdf`.
3. Edward B. Allen, Taghi M. Khoshgoftaar, and Yan Chen. Measuring coupling and cohesion of software modules: an information-theory approach. In *Proceedings Seventh International Software Metrics Symposium*, pages 124–134, 2001.
4. Alexey Bandura, Nikita Kurilenko, Manuel Mazzara, Victor Rivera, Larisa Safina, and Alexander Tchitchigin. Jolie community on the rise. In *9th IEEE International Conference on Service-Oriented Computing and Applications, SOCA*, 2016.
5. Len Bass. *Software architecture in practice*. Pearson Education India, 2007.
6. Len Bass, Paulo Merson, and Liam O'Brien. Quality attributes and service-oriented architectures. *Department of Defense, Technical Report September*, 2005.
7. James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the 1995 Symposium on Software Reusability*, SSR '95, pages 259–262, New York, NY, USA, 1995. ACM.
8. Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *SIGOPS Oper. Syst. Rev.*, 27(5):217–230, 1993.
9. Jan Bosch. Software architecture: The next step. In *Software architecture*, pages 194–199. Springer, 2004.
10. Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Software Composition*, pages 34–50. Springer Berlin Heidelberg, 2007.
11. Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, January 1999.
12. Frederick P Brooks. *The mythical man-month*, volume 1995. Addison-Wesley Reading, MA, 1975.
13. Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, pages 74–95, 2016.
14. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.
15. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Transactions on Programming Languages and Systems*, 34(2):8, 2012.
16. Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, 2016. To appear.
17. Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
18. Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. In *CONCUR*, pages 47–62, 2014.
19. Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. In *CONCUR*, pages 412–426, 2015.
20. Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.
21. Manuel Mazzara Fabrizio Montesi Claudio Guidi, Ivan Lanese. Microservices: a language-based approach. In *Present and Ulterior Software Engineering*. Springer, 2017.
22. Jason Cohen, Eric Brown, Brandon DuRette, and Steven Teleki. *Best kept secrets of peer code review*. Smart Bear, 2006.
23. B Terry Compton and Carol Withrow. Prediction and control of ada software defects. *Journal of Systems and Software*, 12(3):199–207, 1990.
24. Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.
25. Luís Cruz-Filipe and Fabrizio Montesi. Choreographies, computationally. *CoRR*, abs/1510.03271, 2015.
26. Luís Cruz-Filipe and Fabrizio Montesi. Choreographies, divided and conquered. *CoRR*, abs/1602.03729, 2016.
27. Harpal Dhama. Quantitative models of cohesion and coupling in software. *J. Syst. Softw.*, 29(1):65–74, April 1995.
28. N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina. Microservices: How to make your application scale. In *A.P. Ershov Informatics Conference (the PSI Conference Series, 11th edition)*. Springer, 2017.
29. Nicola Dragoni, Schahram Dustdar, Stephan T. Larse, and Manuel Mazzara. Microservices: Migration of a mission critical system. `https://arxiv.org/abs/1704.04173`.
30. Khaled El Emam, N Goel, W Melo, H Lounis, SN Rai, et al. The optimal class size for object-oriented software. *Software Engineering, IEEE Transactions on*, 28(5):494–509, 2002.
31. Michael Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers*, pages 575–607. Springer, 2002.

32. Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
33. Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
34. Martin Fowler and Matthew Foemmel. Continuous integration, 2006. https://www.thoughtworks.com/continuous-integratio
35. Martin Fowler and James Lewis. Microservices, 2014. http://martinfowler.com/articles/microservices.html.
36. Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Self-reconfiguring microservices. In *Theory and Practice of Formal Methods*, pages 194–210. Springer, 2016.
37. Maurizio Gabbrielli, Saverio Giallorenzo, and Fabrizio Montesi. Applied choreographies. *CoRR*, abs/1510.03637, 2015.
38. Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
39. Jerry Gao. A reusable automated acceptance testing architecture for microservices in behavior-driven development. *syntax*, 1(6):7.
40. Jim Gray. A conversation with werner vogels. *ACM Queue*, 4(4):14–22, 2006.
41. William Grosso. *Java RMI*. O'Reilly & Associates, Inc., 1st edition, 2001.
42. Web Services Choreography Working Group et al. Web services choreography description language, 2002.
43. Claudio Guidi. *Formalizing Languages for Service Oriented Computing*. Ph.D. thesis, University of Bologna, 2007.
44. Les Hatton. Reexamining the fault density-component size connection. *IEEE software*, 14(2):89–97, 1997.
45. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.
46. Martin Hitz and Behzad Montazeri. *Measuring coupling and cohesion in object-oriented systems*. Citeseer, 1995.
47. Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. pages 22–138, 1998.
48. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9, 2016. Also: POPL, 2008, pages 273–284.
49. Lucas Krause. *Microservices: Patterns And Applications*. Lucas Krause; 1 edition (April 1, 2015), 2014.
50. V. D. Le, M. M. Neff, R. V. Stewart, R. Kelley, E. Fritzinger, S. M. Dascalu, and F. C. Harris. Microservice-based architecture for the nrdc. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 1659–1664, July 2015.
51. Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, pages 195–211, 2016.
52. Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.
53. Matthew C. MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, Rebekah Metz, and Booz Allen Hamilton. Reference model for service oriented architecture 1.0. *OASIS Standard*, 12, 2006.
54. Tony Mauro. Adopting microservices at netflix: Lessons for team and process design. http://nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/, 2015.
55. Manuel Mazzara. *Towards Abstractions for Web Services Composition*. Ph.D. thesis, University of Bologna, 2006.
56. Manuel Mazzara and Sergio Govoni. *A Case Study of Web Services Orchestration*, pages 1–16. Springer Berlin Heidelberg, 2005.
57. Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.
58. Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.
59. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.
60. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.
61. Bogdan Mingela, Nikolay Troshkov, Manuel Mazzara, Larisa Safina, and Alexander Tchitchigin. Towards static type-checking for jolie. https://arxiv.org/pdf/1702.07146.pdf.
62. Fabrizio Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.
63. Fabrizio Montesi. Process-aware web programming with jolie. *Science of Computer Programming*, 2016.

64. Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-Oriented Programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.
65. Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In *CONCUR*, pages 425–439, 2013.
66. Sam Newman. *Building Microservices*. "O'Reilly Media, Inc.", 2015.
67. Hyacinth S. Nwana. Software agents: an overview. *The Knowledge Engineering Review*, 11:205–244, 9 1996.
68. OASIS. Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.
69. OMG. Common Object Request Broker Architecture. http://www.omg.org/spec/CORBA/.
70. Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319, 2014.
71. Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct 2003.
72. Dewayne E Perry and Alexander L Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
73. Andrew Prunicki. Apache thrift, 2009.
74. Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.
75. Arnon Rotem-Gal-Oz. Fallacies of distributed computing explained. *URL http://www. rgoarchitects. com/Files/fallacies. pdf*, page 20, 2006.
76. Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices (genericity in jolie). In *Proc. of The 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2016*.
77. Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.
78. Yuqiong Sun, Susanta Nanda, and Trent Jaeger. Security-as-a-service for microservices-based cloud applications. In *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, CLOUDCOM '15, pages 50–57, Washington, DC, USA, 2015. IEEE Computer Society.
79. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.
80. Alexander Tchitchigin, Larisa Safina, Manuel Mazzara, Mohamed Elwakil, Fabrizio Montesi, and Victor Rivera. Refinement types in jolie. In *Spring/Summer Young Researchers Colloquium on Software Engineering, SYRCoSE*, 2016.
81. Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 350–369, 2013.
82. Kenton Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 2008.
83. W3C. Web services architecture. http://www.w3.org/TR/ws-arch/.
84. W3C. Web Services Choreography Description Language. https://www.w3.org/TR/ws-cdl-10/.
85. Philip Wadler. Propositions as sessions. 24(2–3):384–418, 2014. Also: ICFP, pages 273–286, 2012.
86. Allen Wang and Sudhir Tonse. Announcing ribbon: Tying the netflix mid-tier services together, January 2013. `http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html`.
87. Zhixian Yan, Manuel Mazzara, Emilia Cimpian, and Alexander Urbanec. Business process modeling: Classifications and perspectives. In *Business Process and Services Computing: 1st International Working Conference on Business Process and Services Computing, BPSC 2007, September 25-26, 2007, Leipzig, Germany.*, page 222, 2007.

Ephemeral Data Handling in Microservices
Technical Report

Saverio Giallorenzo[1], Fabrizio Montesi[1], Larisa Safina[1,2], and Stefano Pio Zingaro[3]

[1]University of Southern Denmark, [2]Innopolis University, [3]Università di Bologna/INRIA

Abstract. In modern application areas for software systems — like eHealth, the Internet-of-Things, and Edge Computing — data is encoded in heterogeneous, tree-shaped data-formats, it must be processed in real-time, and it must be ephemeral, i.e., not persist in the system. While it is preferable to use a query language to express complex data-handling logics, their typical execution engine, a database external from the main application, is unfit in scenarios of ephemeral data-handling. A better option is represented by integrated query frameworks, which benefit from existing development support tools (e.g., syntax and type checkers) and execute within the application memory. In this paper, we propose one such framework that, for the first time, targets tree-shaped, document-oriented queries. We formalise an instantiation of MQuery, a sound variant of the widely-used MongoDB query language, which we implemented in the Jolie language. Jolie programs are microservices, the building blocks of modern software systems. Moreover, since Jolie supports native tree data-structures and automatic management of heterogeneous data-encodings, we can provide a uniform way to use MQuery on any data-format supported by the language. We present a non-trivial use case from eHealth, use it to concretely evaluate our model, and to illustrate our formalism.

1 Introduction

Modern application areas for software systems—like eHealth [1], the Internet of Things [2], and Edge Computing [3]—need to address two requirements: velocity and variety [4]. Velocity concerns managing high throughput and real-time processing of data. Variety means that data might be represented in heterogeneous formats, complicating their aggregation, query, and storage. Recently, in addition to velocity and variety, it has become increasingly important to consider *ephemeral* data handling [5,6], where data must be processed in real-time but not persist — ephemeral data handling can be seen as the opposite of dark data [7], which is data stored but not used. The rise of ephemeral data is due to scenarios with heavy resource constraints (e.g., storage, battery) — as in the Internet of Things and Edge Computing — or new regulations that may limit what data can be persisted, like the GDPR [8] — as in eHealth.

Programming data handling correctly can be time consuming and error-prone with a general-purpose language. Thus, often developers use a query language, paired with an engine to execute them [9]. When choosing the query execution engine, developers can either *A*) use a database management system (DBMS) executed outside of the application, or *B*) include a library that executes queries using the application memory.

Approach *A*) is the most common. Since the early days of the Web, programmers integrated application languages with relational (SQL-based) DBMSs for data persistence and manipulation [10]. This pattern continues nowadays, where relational databases share the scene with new NoSQL [4] DBMSs, like MongoDB [11] and Apache CouchDB [12], which are document-oriented. Document-oriented databases natively support tree-like nested data structures (typically in the JSON format). Since data in modern applications is typically structured as trees (e.g., JSON, XML), this removes the need for error-prone encoding/decoding procedures with table-based structures, as in relational databases. However, when considering ephemeral data handling, the issues of approach *A*) overcome its benefits even if we consider NoSQL DBMSs:

① *Drivers and Maintenance.* An external DBMS is an additional standalone component that needs to be installed, deployed, and maintained. To interact with the DBMS, the developer needs to import in the application specific drivers (libraries, RESTful outlets). As with any software dependency, this exposes the applications to issues of version incompatibility [13].

② *Security Issues.* The companion DBMS is subject to weak security configurations [14] and query injections, increasing the attack surface of the application.

③ *Lack of Tool Support.* Queries to the external DBMS are typically black-box entities (e.g., encoded as plain strings), making them opaque to analysis tools available for the application language (e.g., type checkers) [9].

④ *Decreased Velocity and Unnecessary Persistence.* Integration bottlenecks and overheads degrade the velocity of the system. Bottlenecks derive from resource constraints and slow application-DB interactions; e.g., typical database connection pools [15] represent a potential bottleneck in the context of high data-throughput. Also, data must be inserted in the database and eventually deleted to ensure ephemeral data handling. Overheads also come in the form of data format conversions (see item ⑤).

⑤ *Burden of Variety.* The DBMS typically requires a specific data format for communication, forcing the programmer to develop ad-hoc data transformations to encode/decode data in transit (to insert incoming data and returning/forwarding the result of queries). Implementing these procedures is cumbersome and error-prone.

On the other side, approach *B*) (query engines running within the application) is less well explored, mainly because of the historical bond between query languages and persistent data storage. However, it holds potential for ephemeral data handling. Approach *B*) avoids issues ① and ② by design. Issue ③ is sensibly reduced, since both queries and data can be made part of the application

language. Issue ④ is also tackled by design. There are less resource-dependent bottlenecks and no overhead due to data insertions (there is no DB to populate) or deletions (the data disappears from the system when the process handling it terminates). Data transformation between different formats (item ⑤) is still an issue here since, due to variety, the developer must convert incoming/outgoing data into/from the data format supported by the query engine. Examples of implementations of approach B) are LINQ [16,9] and CQEngine [17]. While LINQ and CQEngine grant good performance (velocity), variety is still an issue. Those proposals either assume an SQL-like query language or rely on a table-like format, which entail continuous, error-prone conversions between their underlying data model and the heterogeneous formats of the incoming/outgoing data.

Contribution. Inspired by approach B), we implemented a framework for ephemeral data handling in microservices; the building blocks of software for our application areas of interest. Our framework includes a query language and an execution engine, to integrate document-oriented queries into the Jolie [18,19] programming language. The language and our implemented framework are open-source projects[1]. Our choice on Jolie comes from the fact that Jolie programs are natively microservices [20]. Moreover, Jolie has been successfully used to build Internet-of-Things [21] and eHealth [22] architectures, as well as Process-Aware Information Systems [23], which makes our work directly applicable to our areas of interest. Finally, Jolie comes with a runtime environment that automatically translates incoming/outgoing data (XML, JSON, etc.) into the native, tree-shaped data values of the language — Jolie values for variables are always trees. By using Jolie, developers do not need to handle data conversion themselves, since it is efficiently managed by the language runtime. Essentially, by being integrated in Jolie, our framework addresses issue ⑤ by supporting *variety by construction*.

As main contribution of this paper, in Section 3, we present the formal model, called TQuery, that we developed to guide the implementation of our Jolie framework. TQuery is inspired by MQuery [24], a sound variant of the MongoDB Aggregation Framework [25]; the most popular query language for NoSQL data handling. The reason behind our formal model is twofold. On the one hand, we abstract away implementation details and reason on the overall semantics of our model — we favoured this top-down approach (from theory to practice) to avoid inconsistent/counter-intuitive query behaviours, which are instead present in the MongoDB Aggregation Framework (see [24] for details). On the other hand, the formalisation is a general reference for implementors; this motivated the balance we kept in TQuery between formal minimality and technical implementation details — e.g., while MQuery adopts a set semantics, we use a tree semantics.

As a second contribution, in Section 2 we present a non-trivial eHealth use case to overview the TQuery operators, by means of their Jolie programming interfaces. The use case is also the first concrete evaluation of MQuery and,

[1] https://github.com/jolie/tquery

Listing 1.1. Snippets of biometric (line 1) and sleep logs (lines 3–5) data.

```
1 [{date:20181129,t:[37,...],hr:[64,...]},
2  {date:20181130,t:[36,...],hr:[66,...]},...]
3
4 [{y:2018,M:[...,{m:11,D:[{d:29,L:[{s:"21:01",e:"22:12",q:"good"}
      ,
5 {s:"22:36",e:"22:58",q:"good"},...]},{d:30,L:[
6 {s:"20:33",e:"22:12",q:"poor"},...]},...]},...]},...]
```

in Section 3, we adopt the use case as our running example to illustrate the semantics of TQuery.

2 A Use Case from eHealth

In this section, we illustrate our proposal with an eHealth use case taken from [26], where the authors delineate a diagnostic algorithm to detect cases of encephalopathy. The handling follows the principle of "data never leave the hospital" in compliance with the GDPR [27]. In the remainder of the paper, we use the use case to illustrate the formal semantics of TQuery. Hence, we do not show here the output of TQuery operators, which are reported in their relative subsections in section 3. While the algorithm described in [26] considers a plethora of clinical tests to signal the presence of the neurological condition, we focus on two early markers for encephalopathy: fever in the last 72 hours and lethargy in the last 48 hours. That data is collectible by commercially-available smart-watches and smart-phones [28]: body temperature and sleep quality. We report in Listing 1.1, in a JSON-like format, code snippets exemplifying the two kinds of data structures. At lines 1–2, we have a snippet of the biometric data collected from the smart-watch of the patient. At lines 4–6 we show a snippet of the sleep logs [29]. Both structures are arrays, marked `[ ]`, containing tree-like elements, marked `{ }`. At lines 1–2, for each `date` we have an array of detected temperatures (`t`) and heart-rates (`hr`). At lines 4–6, to each year (`y`) corresponds an array of monthly (`M`) measures, to a month (`m`), an array of daily (`D`) logs, and to a day (`d`), an array of logs (`L`), each representing a sleep session with its start (`s`), end (`e`) and quality (`q`).

On the data structures above, we define a Jolie microservice, reported in Listing 1.2, which describes the handling of the data and the workflow of the diagnostic algorithm, using our implementation of TQuery. The example is detailed enough to let us illustrate all the operators in TQuery: `match`, `unwind`, `project`, `group`, and `lookup`. Note that, while in Listing 1.2 we hard-code some data (e.g., integers representing dates like `20181128`) for presentation purposes, we would normally use parametrised variables.

In Listing 1.2, line 1 defines a request to an external service, provided by the **`HospitalIT`** infrastructure. The service offers functionality `getPatientPseudoID` which, given some identifying `patientData` (acquired earlier), provides a pseudo-

anonymised identifier — needed to treat sensitive health data — saved in variable `pseudoID`.

At lines 2–6 (and later at lines 9–17) we use the chaining operator `|>` to define a sequence of calls, either to external services, marked by the `@` operator, or to the internal TQuery library. The `|>` operator takes the result of the execution of the expression at its left and passes it as the input of the expression on the right.

At lines 2–6 we use TQuery operators `match` and `project` to extract the recorded temperatures of the patient in the last 3 days/72 hours.

At line 2 we evaluate the content of variable `credentials`, which holds the certificates to let the Hospital IT services access the physiological sensors of a given patient. In the program, `credentials` is passed by the chaining operator at line 3 as the input of the external call to functionality `getMotionAndTemperature` . That service call returns the biometric data (Listing 1.1, lines 1–2) from the **`SmartWatch`** of the patient. While the default syntax of service call in Jolie is the one with the double pair of parenthesis (e.g., at line 1 Listing 1.2), thanks to the chaining operator `|>` we can omit to specify the input of `getMotionAndTemperature` (passed by the `|>` at line 3) and its output (the biometric data exemplified at Listing 1.1) passed to the `|>` at line 4. At line 4 we use the TQuery operator `match` to filter all the entries of the biometric data, keeping only those collected in the last 72 hours/3 days (i.e., since `20181130`). The result of the `match` is then passed to the `project` operator at line 5, which removes all nodes but the temperatures, found under `t` and renamed **`in`** `temperatures` (this is required by the interface of functionality `detectFever`, explained below). The `projection` also includes in its result the `pseudoID` of the patient, **`in`** node `patient_id`. We finally store (line 6) the prepared data in variable `temps` (since it will be used both at line 7 and 16).

At line 7, we call the external functionality `detectFever` to analyse the temperatures and check if the patient manifested any fever, storing the result in variable `hasFever`.

After the analysis on the temperatures, **`if`** the patient `hasFever` (line 8), we continue testing for lethargy. To do that, at lines 9–10, we follow the same strategy described for lines 2–3 to pass the `credentials` to functionality `getSleepPatterns` , used to collect the sleep logs of the patient from her **`SmartPhone`**. Since the sleep logs are nested under years, months, and days, to filter the logs relative to the last 48 hours/2 days, we first flatten the structure through the `unwind` operator applied on nodes `M.D.L` (line 11). For each nested node, separated by the dot (`.`), the `unwind` generates a new data structure for each element in the array reached by that node. Concretely, the array returned by the `unwind` operator at line 11 contains all the sleep logs in the shape: `[ {year:2018,M:[{m:11,D:[{d:29,L:[{s:"21:01",e:"22:12",q:"good"}]}]}]}`

`{year:2018,M:[{m:11,D:[{d:29,L:[{s:"22:36",e:"22:58",q:"good"}]}]}]} ]`

where there are as many elements as there are sleep logs and the arrays under `M`, `D`, and `L` contain only one sleep log. Once flattened, at line 12 we modify the data-structure with the `project` operator to simplify the subsequent chained commands: we rename the node `y` **`in`** `year`, we move and rename the node `M.m`

Listing 1.2. Encephalopathy Diagnostic Algorithm.

```
1  getPatientPseudoID@HospitalIT( patientData )( pseudoID );
2  credentials
3  |> getMotionAndTemperature@SmartWatch
4  |> match {date == 20181128 || date == 20181129 || date == 20181130
       }
5  |> project {t in temperatures, pseudoID in patient_id }
6  |> temps;
7  detectFever@HospitalIT( temps )( hasFever );
8  if( hasFever ){
9    credentials
10   |> getSleepPatterns@SmartPhone
11   |> unwind {M.D.L }
12   |> project{y in year,M.m in month,M.D.d in day,M.D.L.q in
         quality}
13   |> match {year == 2018 && month == 11 && ( day == 29 || day == 30
         ) }
14   |> group { quality by day, month, year }
15   |> project {quality, pseudoID in patient_id }
16   |> lookup {patient_id == temps.patient_id in temps }
17   |> detectEncephalopathy@HospitalIT }
```

`in month` (bringing it at the same nesting level of `year`); similarly, we move `M.D.d`, renaming it `day`, and we move `M.D.L.q` (the log the quality of the sleep), renaming it `quality` — `M.D.L.s` and `M.D.L.e`, not included in the `project`, are discarded. On the obtained structure, we filter the sleep logs relative to the last 48 hours with the `match` operator at line 13. At line 14 we use the `group` operator to aggregate the `quality` of the sleep sessions recorded in the same day (i.e., grouping them **`by`** `day`, `month`, and `year`). Finally, at line 15 we select, through a `projection`, only the aggregated values of `quality` (getting rid of `day`, `month`, and `year`) and we include under node `patient_id` the `pseudoID` of the patient. That value is used at line 16 to join, with the `lookup` operator, the obtained sleep logs with the previous values of temperatures (`temps`). The resulting, merged data-structure is finally passed to the **`HospitalIT`** services by calling the functionality `detectEncephalopathy`.

3 TQuery Framework

In this section, we define the formal syntax and semantics of the operators of TQuery. We begin by defining data trees:

$$\mathsf{T} \ni \mathsf{t} ::= \mathsf{b}\ \{\mathsf{k}_i : \mathsf{a}_i\}_i \qquad\qquad \mathsf{A} \ni \mathsf{a} ::= [\mathsf{t}_1, \cdots, \mathsf{t}_n]$$

Above, each tree $\mathsf{t} \in \mathsf{T}$ has two elements. First, a *root* value b, $\mathsf{b} \in \mathtt{sort}$, where $\mathtt{sort} = \mathtt{str} \cup \mathtt{int} \cup \cdots \cup \{\upsilon\}$ and υ is the null value. Second, a set of one-dimensional vectors, or arrays, containing sub-trees. Each array is identified by a label $\mathsf{k} \in \mathsf{K}$.

We write arrays $a \in A$ using the standard notation $[t_1, \cdots, t_n]$. We write $k(t)$ to indicate the extraction of the array pointed by label k in t: if k is present in t we return the related array, otherwise we return the null array α, formally

$$k(\ b\ \{k_i : a_i\}_i\ ) = \begin{cases} a & \text{if } (k : a) \in \{k_i : a_i\}_i \\ \alpha & \text{otherwise} \end{cases}$$

We assume the range of arrays to run from the minimum index 1 to the maximum $\#a$, which we also use to represent the size of the array. We use the standard index notation $a[i]$ to indicate the extraction of the tree at index i in array a. If a contains an element at index i we return it, otherwise we return the null tree τ.

$$a[i] = \begin{cases} t_i & \text{if } a = [t_1, \cdots, t_n] \ \wedge \ 1 \leq i \leq n \\ \tau & \text{otherwise} \end{cases}$$

Example 1 (Data Structures). To exemplify our notion of trees, we model the data structures in Listing 1.1.

```
1 [ υ {date:[20181129 {}],t:[37{},...],hr:[64{},...]},
2   υ {date:[20181130 {}],t:[36{},...],hr:[66{},...]},...]
3
4 [υ{y:[2018{}],M:[υ{m:[11{}],D:[
5   υ{d:[29{}],L:[υ{s:["21:01"{}],e:["22:12"{}],q:["good"{}]},
        ...]},
6   υ{d:[30{}],L:[υ{s:["20:33"{}],e:["22:12"{}],q:["poor"{}]},
        ...]},
7 ...]},...]},...]
```

Note that tree roots hold the values in the data structure (e.g., the integer representation of the date 20181128). When root values are absent, we use the null value υ.

We define paths $p \in P$ to express tree traversal: $P \ni p ::= e\ .\ p\ \mid\ \epsilon$. Paths are concatenations of expressions e, each assumed to evaluate to a tree-label, and the sequence termination ϵ (often omitted in examples). The application of a path p to a tree t, written $[\![\, t \,]\!]^p$ returns an array that contains the sub-trees reached traversing t following p. This is aligned with the behaviour of path application in MQuery which return a set of trees. In the reminder of the paper, we write $e{\downarrow}k$ to indicate that the evaluation of expression e in a path results into the label k. Also, both here and in MQuery paths neglect array indexes: for a given path $e.p$, such that $e{\downarrow}k$, we apply the subpath p to all trees pointed by k in t. We use the standard array concatenation operator $::$ where $[t_1, \cdots, t_n] = [t_1] :: \cdots :: [t_n]$. We can finally define $[\![\, p \,]\!]^t$, which either returns an array of trees or the null array α in case the path is not applicable.

$$[\![\, p \,]\!]^t = \begin{cases} [\![\, p' \,]\!]^{t_1} \ :: \ \cdots \ :: \ [\![\, p' \,]\!]^{t_n} & \text{if } p = e.p' \ \wedge \ e{\downarrow}k \ \wedge \ k(t) = [t_1, \cdots, t_n] \\ [\ t\ ] & \text{if } p = \epsilon \\ \alpha & \text{otherwise} \end{cases}$$

In the reminder, we also assume the following structural equivalences

$$\alpha :: \alpha \equiv \alpha \qquad \alpha :: [] \equiv [] :: \alpha \equiv [] :: [] \equiv [] \qquad \alpha :: a \equiv a :: \alpha \equiv [] :: a \equiv a :: [] \equiv a$$

Example 2. Let us see some examples of path-tree application where we assume a tree $t = v$ `{x: [v {z: [1 {}, 2 {} ] , y: [3{}] } ] }`

```
1 ⟦x.ϵ⟧t  ⇒ [v{z:[1{},2{}],y:[3{}]}]
2 ⟦x.z.ϵ⟧t ⇒ [1{},2{}]
```

We first present the syntax of TQuery and then dedicate a subsection to the semantics of each operator and to the running examples that illustrate its behaviour.

As in MQuery, a TQuery query is a sequence of stages s applied on an array a: $a \triangleright s \cdots \triangleright s$. The staging operator $\triangleright$ in TQuery is similar to the Jolie chaining operator `|>`: they evaluate the expression on their left, passing its result as input to the expression at their right. We report in Figure 1 the syntax of TQuery, which counts five stages. The *match* operator μ_φ selects trees according to the criterion φ. Such criterion is either the boolean truth `true`, a condition expressing the equality of the application of path p and the array a, a condition expressing the equality of the application of path p_1 and the application of a second path p_2, the existence of a path $\exists p$, and the standard logic connectives negation $\neg$, conjunction $\wedge$, and disjunction $\vee$. The *unwind* operator ω_p flattens an array reached through a path p and outputs a tree for each element of the array. The *project* operator π_Π modifies trees by projecting away paths, renaming paths, or introducing new paths, as described in the sequence of elements in Π, which are either a path p or a value definition d inserted into a path p. Value definitions are either: a boolean value b (`true` or `false`), the application of a path p, an array of value definitions, a criterion φ or the ternary expression, which, depending the satisfiability of criterion φ selects either value definition d_1 or d_2. The *group* operator $\gamma_{\Gamma,\Gamma'}$ groups trees according to a grouping condition Γ and aggregates values of interest according to Γ'. Both Γ and Γ' are sequences of elements of the form $p \rangle p'$ where p is a path in the input trees, and p' a path in the output trees. The *lookup* operator $\lambda_{q=a.r \rangle p}$ joins input trees with trees in an external array a. The trees to be joined are found by matching those input trees whose array found applying path q equals the ones found applying path r to the trees of the external array a. The matching trees from a are stored in the matching input trees under path p.

3.1 Match

When applied to an array a, match μ_φ returns those elements in a that satisfy φ. If there is no element in a that satisfies φ, μ_φ returns an array with no elements (different from α). Below, we mark $t \models \varphi$ the satisfiability of criterion

$$
\begin{aligned}
s &::= \mu_\varphi \mid \omega_p \mid \pi_\Pi \mid \gamma_{\Gamma:\Gamma'} \mid \lambda_{q=a.r\rangle p} \\
\varphi &::= \mathsf{true} \mid p = a \mid p_1 = p_2 \mid \exists p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
\Pi &::= p \mid d\rangle p \mid p, \Pi \mid d\rangle p, \Pi \\
d &::= b \mid p \mid [d_1, \cdots, d_n] \mid \varphi \mid \varphi ? d_1 : d_2 \\
\Gamma &::= p\rangle p' \mid p\rangle p', \Gamma
\end{aligned}
$$

Fig. 1. Syntax of the TQuery

φ by a tree t.

$$
\alpha \triangleright \mu_\varphi = [\,] \qquad [t] :: a \triangleright \mu_\varphi = \begin{cases} [t] :: (a \triangleright \mu_\varphi) & \text{if } t \models \varphi \\ a \triangleright \mu_\varphi & \text{if } \#a > 0 \\ [\,] & \text{otherwise} \end{cases}
$$

$$
t \models \varphi \text{ holds iff } \begin{cases} \varphi = \mathsf{true} \\ \varphi = (\exists p) \ \wedge \ [\![p]\!]^t \neq \alpha \\ \varphi = (p = a) \ \wedge \ [\![p]\!]^t = a \\ \varphi = (p_1 = p_2) \ \wedge \ t \models \big((p_1 = a) \wedge (p_2 = a)\big) \\ \varphi = (\neg\varphi') \ \wedge \ t \not\models \varphi' \\ \varphi = (\varphi_1 \wedge \varphi_2) \ \wedge \ (t \models \varphi_1 \ \wedge \ t \models \varphi_2) \\ \varphi = (\varphi_1 \vee \varphi_2) \ \wedge \ (t \models \varphi_1 \ \vee \ t \models \varphi_2) \end{cases}
$$

Above, criterion $\varphi = (p_1 = p_2)$ is satisfied both when the application of the two paths to the input tree t return the same array a as well as when both paths do not exist in t, i.e., their application coincide on α.

Example 3. We report below the execution of the `match` operator at line 4 of Listing 1.2. In the example, array a corresponds to the data structure defined at lines 1–2 in Example 1. First we formalise in TQuery the `match` operator at line 4: $a \triangleright \mu_\varphi$ where

$\varphi =$ `date == [20181128{}] ∨ date == [20181129{}] ∨ date == [20181130{}]`

The match evaluates all trees inside a, below we just show that evaluation for $a[1]$

```
1  a[1] = υ {date:[20181129{}],t:[37{},...],hr:[ 64{},...]}
```

we verify if one of the sub-conditions $a[1] \models$ `date == [20181128{}]`, $a[1] \models$ `date == [20181129{}]`, or $a[1] \models$ `date == [20181130{}]` hold. Each condition is evaluated by applying path `date` on $a[1]$ and by verifying if the equality with the considered array, e.g., `[20181128{}]`, holds. As a result, we obtain the input array a filtered from the trees that do not correspond to the dates in the criterion.

3.2 Unwind

To define the semantics of the unwind operator ω, we introduce the *unwind expansion operator* $\mathbf{E}(t, a)^k$ (read "unwind t on a under k"). Informally $\mathbf{E}(t, a)^k$ returns an array of trees with cardinality $\#a$ where each element has the shape of t except that label k is associated index-wise with the corresponding element in a. Formally, given a tree t, an array a, and a key k:

$$\mathbf{E}(t, a)^k = \begin{cases} \left[b\Big((\{k_i : a_i\}_i \setminus \{k : k(t)\}) \cup \{k : [t']\} \Big) \right] :: \mathbf{E}(t, a')^k & \text{if } \begin{array}{l} a = [t'] :: a' \\ \wedge\ t = b\{k_i : a_i\}_i \end{array} \\ [\,] & \text{otherwise} \end{cases}$$

Then, the formal definition of $a \triangleright \omega_p$ is

$$a \triangleright \omega_p \begin{cases} \mathbf{E}(t, [\![k.\epsilon]\!]^t \triangleright \omega_{p'})^k :: a' \triangleright \omega_p & \text{if } p = e.p' \ \wedge\ e{\downarrow}k \ \wedge\ a = [t] :: a' \\ a & \text{if } p = \epsilon \\ [\,] & \text{otherwise} \end{cases}$$

We define the unwind operator inductively over both a and p. The induction over a results in the application of the unwind expansion operator $\mathbf{E}$ over all elements of a. The induction over p splits p in the current key k and the continuation p'. Key k is used to retrieve the array in the current element of a, i.e., $[\![k.\epsilon]\!]^t$, on which we apply ω'_p to continue the unwind application until we reach the termination with $p = \epsilon$.

Example 4. We report the execution of the `unwind` operator at line 11 of Listing 1.2.

The unwind operator unfolds the given input array wrt a given path p in two directions. The first is breadth, where we apply the unwind expansion operator $\mathbf{E}(t, a')^k$, over all input trees t and wrt the first node k in the path p. The second direction is depth, and defines the content of array a' in $\mathbf{E}(t, a')^k$, which is found by recursively applying the unwind operator wrt to the remaining path nodes in p (k excluded) over the arrays pointed by node k in each t.

Let a be the sleep-logs data-structure at lines 4–6 of Example 1, such that $a = [t_{2018}, t_{2017}, ...]$ where e.g., t_{2018} is that t in a such that $[\![y]\!]^t = [2018\{\}]$. The concatenation below is the first level of depth unfolding, i.e., for node `M` of unwind $\omega_{M.D.L}$.

$$\mathbf{E}(t_{2018}, [\![M.\epsilon]\!]^{t_{2018}} \triangleright \omega_{D.L})^M :: \mathbf{E}(t_{2017}, [\![M.\epsilon]\!]^{t_{2017}} \triangleright \omega_{D.L})^M :: ...$$

To conclude this example, we show the execution of the unwind expansion operator $\mathbf{E}(t_{30}, [\![L.\epsilon]\!]^{t_{30}})^L$ of the terminal node `L` in path p, relative to the sleep logs recorded within a day, represented by tree t_{30}, i.e., where $[\![t_{30}]\!]^d = [30\{\}]$.

$\mathbf{E}(t_{30}, [\![L.\epsilon]\!]^{t_{30}})^L \Rightarrow$
[υ ({d : [30{}], L : [...]} \ {L : [...]}) ∪ {L : [υ{s:["21:01"{}], e:["22:12"{}], q:["good"{}]}]})] ::
[υ ({d : [30{}], L : [...]} \ {L : [...]}) ∪ {L : [υ{s:["22:36"{}], e:["22:58"{}], q:["good"{}]}]})] :: ...

Above, for each element of the array pointed by `L`, e.g., `{s:["21:01"{}], e:["22:12"{}], q:["good"{}]}` we create a new structure [υ ({d : [30{}], L : [...]}]

where we replace the original array associated with the key `L` with a new array containing only that element. The final result of the `unwind` operator has the shape:

```
[υ {y:[2018{}],M:[υ{m:[11{}],D:[υ {d:[30{}],
      L:[υ{s:"21:01",e:"22:12",q:"good"}]}]}]},
 υ {y:[2018{}],M:[υ{m:[11{}],D:[υ {d:[30{}],
      L:[υ{s:"22:36",e:"22:58",q:"good"}]}]}]},... ]
```

3.3 Project

We start by defining some auxiliary operators used in the definition of the project. Auxiliary operators $\pi_p(a)$ and $\pi_p(t)$ formalise the application of a branch-selection over a path p. Then, the auxiliary operator $\mathbf{eval}(d, t)$ returns the array resulting from the evaluation of a definition d over a tree t. Finally, we define the projection of a value (definition) d into a path p over a tree t i.e., $\pi_{d\rangle p}(t)$. The projection for a path p over an array a results in an array where we project p over all the elements (trees) of a.

$$\pi_p([t_1, \cdots, t_n]) = [\ \pi_p(t_1), \cdots, \pi_p(t_n)\ ]$$

The projection for a path p over a tree t implements the actual semantics of branch-selection, where, given a path $e.p'$, $e{\downarrow}k$, we remove all the branches k_i in $t = b\{k_i : a_i\}$, keeping only k (if $k \in \{k_i\}$) and continue to apply the projection for the continuation p' over the (array of) sub-trees under k in t (i.e., $[\![\, k.\epsilon \,]\!]^t$).

$$\pi_p(t) = \begin{cases} \upsilon\ \{\ k : \pi_{p'}([\![\, k.\epsilon \,]\!]^t)\ \} & \text{if } [\![\, p \,]\!]^t \neq \alpha\ \wedge\ p = e.p'\ \wedge\ t = b\ \{k_i : a_i\}_i\ \wedge\ e{\downarrow}k \\ t & \text{if } p = \epsilon \\ \tau & \text{otherwise} \end{cases}$$

The operator $\mathbf{eval}(d, t)$ evaluates the value definition d over the tree t and returns an array containing the result of the evaluation.

$$\mathbf{eval}(d, t) = \begin{cases} [\ d\ \{\}\ ] & \text{if } d \in V \\ [\ t \models \varphi\ \{\}\ ] & \text{if } d \in \varphi \\ [\![\, d \,]\!]^t & \text{if } d \in P \\ \mathbf{eval}(d, t) :: \mathbf{eval}(d', t) & \text{if } d = [d] :: d' \\ \mathbf{eval}(d', t) & \text{if } d = \varphi ? d_{\texttt{true}} : d_{\texttt{false}}\ \wedge\ d' = d_{t \models \varphi} \\ \alpha & \text{otherwise} \end{cases}$$

Then, the application of the projection of a value definition d on a path p, i.e., $\pi_{d\rangle p}(t)$ returns a tree where under path p is inserted the **eval**uation of d over t.

$$\pi_{d\rangle p}(t) = \begin{cases} \upsilon\ \{k : [\ \pi_{d\rangle p'}(t)\ ]\} & \text{if } p = e.p'\ \wedge e{\downarrow}k\ \wedge\ \mathbf{eval}(d, t) \neq \alpha \\ \upsilon\ \{k : \mathbf{eval}(d, t)\} & \text{if } p = e.\epsilon\ \ \wedge e{\downarrow}k\ \wedge\ \mathbf{eval}(d, t) \neq \alpha \\ \tau & \text{otherwise} \end{cases}$$

Before formalising the projection, we define the auxiliary tree-merge operator $t \oplus t'$, used to merge the result of a sequence of projections Π.

$$([t] :: a) \oplus ([t'] :: a') = [t \oplus t'] :: a \oplus a' \qquad a \oplus [] = [] \oplus a = a \oplus \alpha = \alpha \oplus a = a$$

$$b\ \{k_i : a_i\}_i \oplus b'\ \{k_j : a_j\}_j = \tau \quad \text{if } b \neq b' \qquad t \oplus \tau = t$$

$$t \oplus t' = b\ \{\ k_h : k_h(t) \oplus k_h(t')\ \}_{h \in I \cup J} \quad \text{if } t = b\ \{\ k_i : a_i\ \}_{i \in I}\ \wedge\ t' = b\ \{\ k_j : a_j\ \}_{j \in J}$$

To conclude, first we define the application of the projection to a tree t, i.e., $t \triangleright \pi_\Pi$, which merges ($\oplus$) into a single tree the result of the applications of projections Π over t

$$\pi_\Pi(t) = \begin{cases} \pi_p(t) \oplus (t \triangleright \pi_{\Pi'}) & \text{if } \Pi = p, \Pi' \\ \pi_{d\rangle p}(t) \oplus (t \triangleright \pi_{\Pi'}) & \text{if } \Pi = d\rangle p, \Pi' \\ \pi_p(t) & \text{if } \Pi = p \\ \pi_{d\rangle p}(t) & \text{if } \Pi = d\rangle p \end{cases}$$

and finally, we define the application of the projection π_Π to an array a, i.e., $a \triangleright \pi_\Pi$, which corresponds to the application of the projection to all the elements of a.

$$[\ t_1, \cdots, t_n\ ] \triangleright \pi_\Pi = [\ \pi_\Pi(t_1),\ \cdots\ , \pi_\Pi(t_n)\ ]$$

Example 5. We report the execution of the `project` at line 5 of Listing 1.2. Let a be the array at the end of Example 3, and let t_{28}, t_{29}, t_{30} be the trees in a such that t_{28} is the first tree in a relative to `date 20181128`, t_{29} the second, and t_{30} the third

$$[t_{28}, t_{29}, t_{29}] \ \triangleright\ \pi_{t\rangle \text{temperatures, pseudoID}\rangle \text{patient\_id}} \Rightarrow$$
$$[\pi_{t\rangle \text{temperatures, pseudoID}\rangle \text{patient\_id}}(t_{28}), \pi_{t\rangle \text{temperatures, pseudoID}\rangle \text{patient\_id}}(t_{29}), \pi_{t\rangle \text{temperatures, pseudoID}\rangle \text{patient\_id}}(t_{30})]$$

We continue showing the projection of the first element in a, t_{28} (the projection on the other elements follows the same structure)

$$\pi_{t\rangle \text{temperatures, pseudoID}\rangle \text{patient\_id}}(t_{28}) \Rightarrow \pi_{t\rangle \text{temperatures}}(t_{28}) \oplus \pi_{\text{pseudoID}\rangle \text{patient\_id}}(t_{28}) \Rightarrow$$
$$v\ \{\text{temperatures} : \pi_{t\rangle \epsilon}(t_{28})\ \} \oplus v\ \{\text{patient\_id} : \pi_{[\text{"xxx"}\{\}]\rangle \epsilon}(t_{28})\ \}$$
$$= v\ \{\text{temperatures} : \textbf{eval}(t, t_{28})\ \} \oplus v\ \{\text{patient\_id} : \textbf{eval}([\text{"xxx"}\{\}], t_{28})\ \}$$
$$= v\ \{\text{temperatures} : [\![t]\!]^{t_{28}}\ \} \oplus \{\text{patient\_id: ["xxx"\{\}]}\}$$
$$= v\ \{\text{temperatures} : [36\{\}, \ldots], \text{patient\_id: ["xxx"\{\}]}\ \}$$

The result of the projection has the shape

```
[v {temperatures:[36{},...], patient_id:["xxx"{}] },
 v {temperatures:[37{},...], patient_id:["xxx"{}] },
 v {temperatures:[36{},...], patient_id:["xxx"{}] ]
```

Example 6. We report the execution of the `project` at line 12 of Listing 1.2. Let a be the array at the end of Example 4, and let t^1_{2018}, t^2_{2018}, ... be the trees in a such that t^1_{2018} is the first tree in a relative to `year 2018`, t^2_{2018} the second, and so on.

$$[t^1_{2018}, t^2_{2018}, \ldots] \triangleright \pi_{y\rangle year,\ M.m\rangle month,\ M.D.d\rangle day,\ M.D.L.q\rangle quality} \Rightarrow$$
$$[\pi_{y\rangle year,\ M.m\rangle month,\ M.D.d\rangle day,\ M.D.L.q\rangle quality}(t^1_{2018}), \pi_{y\rangle year,\ M.m\rangle month,\ M.D.d\rangle day,\ M.D.L.q\rangle quality}(t^2_{2018}), \ldots]$$

We continue showing the projection of the first element in a, t^1_{2018} (the projection on the other elements follows the same structure)

$$\pi_{y\rangle year,\ M.m\rangle month,\ M.D.d\rangle day,\ M.D.L.q\rangle quality}(t^1_{2018}) \Rightarrow$$
$$\pi_{y\rangle year}(t^1_{2018}) \oplus \pi_{M.m\rangle month}(t^1_{2018}) \oplus \pi_{M.D.d\rangle day}(t^1_{2018}) \oplus \pi_{M.D.L.q\rangle quality}(t^1_{2018})$$

Finally, we show the unfolding of the first two projections from the left, above, i.e., those for $y\rangle year$ and for $M.m\rangle month$, and their merge $\oplus$ (the remaining ones unfold similarly).

$$\pi_{y\rangle year}(t^1_{2018}) \oplus \pi_{M.m\rangle month}(t^1_{2018}) \Rightarrow$$
$$\upsilon\ \{year\ :\ \pi_{y\rangle \epsilon}(t^1_{2018})\ \} \oplus \upsilon\ \{month\ :\ \pi_{M.m\rangle \epsilon}(t^1_{2018})\ \}$$
$$= \upsilon\ \{year\ :\ \mathbf{eval}(y, t^1_{2018})\ \} \oplus \upsilon\ \{month\ :\ \mathbf{eval}(M.m, t^1_{2018})\ \}$$
$$= \upsilon\ \{year\ :\ [\![y]\!]^{t^1_{2018}}\ \} \oplus \upsilon\ \{month\ :\ [\![M.m]\!]^{t^1_{2018}}\ \}$$
$$= \upsilon\ \{year\ :\ [2018\{\}]\ \} \oplus \upsilon\ \{month\ :[11\{\}]\ \}$$
$$= \upsilon\ \{year\ :\ [2018\{\}],\ month\ :\ [11\{\}]\ \}$$

The result of the projection has the shape

```
[υ{year:[2018{}],month:[11{}],day:[30{}],quality:["good"{}]},
 υ{year:[2018{}],month:[11{}],day:[30{}],quality:["good"{}]},
 υ{year:[2018{}],month:[11{}],day:[30{}],quality:["poor"{}]},
     ...]
```

3.4 Group

The group operator takes as parameters two sequences of paths, separated by a semicolon, i.e., $q_1\rangle p_1, \cdots, q_n\rangle p_n : s_1\rangle r_1, \cdots, s_m\rangle r_m$. The first sequence of paths, ranged $[1, n]$, is called *aggregation set*, while the second sequence, ranged $[1, m]$, is called *grouping set*. Intuitively, the group operator first groups together the trees in a which have the maximal number of paths $s_1, \cdots, s_m$ in the grouping set whose values coincide. The values in $s_1, \cdots, s_m$ are projected in the corresponding paths $r_1, \cdots, r_m$. Once the trees are grouped, the operator aggregates all the different values, without duplicates, found in paths $q_1, \cdots, q_n$ from the aggregation set, projecting them into the corresponding paths $p_1, \cdots, p_n$. We start the definition of the grouping operator by expanding its application to an array a. In the expansion below, on the right, we use the series-concatenation operator $\vdots$ and the set H, element of the power set $2^{[1,n]}$, to range over all possible combinations of paths in the grouping set. Namely, the expansion corresponds to the concatenation of all the arrays resulting from the application of the group operator on a subset (including the empty and the whole set) of paths in the grouping set.

$$\gamma_{q_1\rangle p_1, \cdots, q_n\rangle p_n : s_1\rangle r_1, \cdots, s_m\rangle r_m} \triangleright a = \vdots_{\forall H \in 2^{[1,m]}} \gamma^H_{q_1\rangle p_1, \cdots, q_n\rangle p_n : s_1\rangle r_1, \cdots, s_m\rangle r_m}(a)$$

In the definition of the expansion, we mark $\{\!\{a\}\!\}$ the casting of an array a to a set (i.e., we keep only unique elements in a and lose their relative order). Each $\gamma^{H}_{q_1\rangle p_1,\cdots,q_n\rangle p_n:s_1\rangle r_1,\cdots,s_m\rangle r_m}(a)$ returns an array that contains those trees in a that correspond to the grouping illustrated above. Formally:

$$\gamma^{H}_{q_1\rangle p_1,\cdots,q_n\rangle p_n:s_1\rangle r_1,\cdots,s_m\rangle r_m}(a) =$$

$$\begin{cases} \begin{matrix}\bullet\bullet\\ \forall a'\end{matrix}\left[\bigoplus_{i=1}^{n}\pi_{\chi,\theta_i}(\tau)\right] & \begin{aligned} & h\in H \ \wedge\ a'[h]\in\left\{[\![\, s_h\,]\!]^t \mid t\in\{\!\{a\}\!\}\wedge[\![\, s_h\,]\!]^t\neq\alpha\right\}\\ & \wedge\,\chi=(a'[h]\rangle H,[r_1,\cdots,r_n])\\ \text{if } & \wedge\,\theta_i=\begin{matrix}\bullet\bullet\\ \forall t_i\end{matrix}[\![\, q_i\,]\!]^{t_i}\rangle p_i \ \wedge\ t_i\in\{\!\{a\triangleright\mu_{\psi_i}\}\!\}\supset\varnothing\\ & \wedge\,\psi_i=\exists q_i \ \wedge\ \neg\bigvee_{j\notin H}\exists s_j \ \wedge\bigwedge_{h}\Big(\big(s_h=a'[h]\big)\wedge\exists s_h\ \Big)\end{aligned}\\ [\,] & \text{otherwise}\end{cases}$$

When applied over a set H, $h\in H$, γ considers *all* combinations of values identified by paths s_h in the trees in a. In the formula above, we use the array a' to refer to those combinations of values. In the definition, we impose that, for each element in a' in a position h, there must be at least one tree in a that has a non-null ($\neq\alpha$) array under path s_h. Hence, for each combination a' of values in a, γ builds a tree that *i)* contains under paths r_h the value $a'[h]$ (as encoded in the projection query χ and from the definition of the operator $a'[h]\rangle H,a$, defined below) and *ii)* contains under paths p_i, $1\leq i\leq n$, the array containing all the values found under the correspondent path q_i in all trees in a that match the same combination element-path in a' (as encoded in θ_i). The grouping is valid (as encoded in ψ_i) only if we can find (i.e., match μ) trees in a where *i)* we have a non-empty value for q_i, *ii)* there are no paths s_j that are excluded in H, and *iii)* for all paths considered in H, the value found under path s_h corresponds to the value in the considered combination $a'[h]$. If the previous conditions are not met, γ returns an empty array $[\,]$.

We conclude defining the operator $a'[h]\rangle H,a$, used above to unfold the set of aggregation paths and the related values contained in H, e.g., let $H=\{1,3,5\}$ then $a'[h]\rangle H,a=a'[1]\rangle p_1,a'[3]\rangle p_3,a'[5]\rangle p_5$. Its meaning is that, for each path $p_\bullet$, we project in it the value correspondent to $a'[\bullet]$. Formally

$$a'[h]\rangle H,a=\begin{cases}a'[j]\rangle a[j],(a'[h]\rangle(H\setminus\{j\}),a) & \text{if } |H|>1\wedge j\in H\\ a'[j]\rangle a[j] & \text{if } |H|=1\wedge j\in H\\ \epsilon & \text{otherwise}\end{cases}$$

Note that for case $\gamma^{\varnothing}_{\cdots}$ (i.e., for $H=\varnothing$), $a'[h]\rangle H,a$ returns the empty path ϵ, which has no effect (i.e., it projects the input tree) in the projection π_χ in the definition of γ. Hence, the resulting tree from grouping over $\varnothing$ will just include (and project over $p_1,\cdots,p_n$) those trees in a that do not include any value reachable by paths $s_1,\cdots,s_m$ (as indicated by expression $\neg\bigvee_{j\notin H}\exists s_j$ in ψ_i).

Like in MQuery and MongoDB, we allow the omission of paths $p_1,\cdots,p_n$ and $r_1,\cdots,r_n$ in $\Gamma:\Gamma'$. However, we interpret this omission differently wrt

MQuery. There, the values obtained from q_is with missing p_is (resp., s_i with missing r_i) are stored within a default path `_id`. Here, we intend the omission as an indication of the fact that the user wants to preserve the structure of q_i (resp., s_i) captured by the structural equivalence below.

$$\gamma_{q_1,\cdots,q_n:s_1,\cdots,s_m} \equiv \gamma_{q_1\rangle q_1,\cdots,q_n\rangle q_n:s_1\rangle s_1,\cdots,s_m\rangle s_m}$$

Example 7. We report the execution of the `group` operator at line 14 of Listing 1.2. Let a be the result of the projection Example 6, with the exception that a has been filtered by the `match` at line 13 in Listing 1.2 and contains only the sleep logs for `days` 29 and 30 of `month` 11 and `year` 2018.

$$a \triangleright \gamma_{\texttt{quality : day,month,year}} \equiv a \triangleright \gamma_{\texttt{quality}\rangle\texttt{quality : day}\rangle\texttt{day,month}\rangle\texttt{month,year}\rangle\texttt{year}}$$

$$\Rightarrow \vdots_{\forall H \in 2^{[1,1]}} \gamma^{H}_{\texttt{quality}\rangle\texttt{quality : day}\rangle\texttt{day,month}\rangle\texttt{month,year}\rangle\texttt{year}}(a)$$

$$= \left[\gamma^{\varnothing}_{\texttt{quality}\rangle\texttt{quality : day}\rangle\texttt{day,month}\rangle\texttt{month,year}\rangle\texttt{year}}(a)\right] \quad \text{this to equals } [\,] \text{ since } \psi_i \text{ is always false in } a$$

$$:: \left[\gamma^{\{1\}}_{\texttt{quality}\rangle\texttt{quality : day}\rangle\texttt{day,month}\rangle\texttt{month,year}\rangle\texttt{year}}(a)\right]$$

$$= [\,] :: \left[\pi_{\texttt{[30 \{\}]}\rangle\texttt{day},\texttt{[11 \{\}]}\rangle\texttt{month},\texttt{[2018 \{\}]}\rangle\texttt{year}}(\tau) \oplus \pi_{\texttt{quality}\rangle\texttt{["good"\{\},"good"\{\},...]}}(\tau)\right]$$

$$:: \left[\pi_{\texttt{[29 \{\}]}\rangle\texttt{day},\texttt{[11 \{\}]}\rangle\texttt{month},\texttt{[2018 \{\}]}\rangle\texttt{year}}(\tau) \oplus \pi_{\texttt{quality}\rangle\texttt{["good"\{\},"poor"\{\},...]}}(\tau)\right]$$

$$= \texttt{[}\upsilon\texttt{\{ day: [30\{\}],month:[11\{\}],year:[2018\{\}],quality:["good"\{\},"good"\{\},...]},}$$
$$\upsilon\texttt{\{ day: [29\{\}],month:[11\{\}],year:[2018\{\}],quality:["good"\{\},"poor"\{\},...]\}]}$$

3.5 Lookup

Informally, the lookup operator joins two arrays, a source a and an adjunct a', wrt a destination path p and two source paths q and r. Result of the lookup is a new array that has the shape of the source array a but where each of its elements t has under path p those elements in the adjunct array a' whose values under path r equal the values found in t under path q. Formally

$$a \triangleright \lambda_{q=a'.r\rangle p} = [\,\pi_{\epsilon,\beta_1}(a[1])\,] :: \cdots :: [\,\pi_{\epsilon,\beta_n}(a[n])\,] \text{ s.t. } \begin{cases} \beta_i = (a' \triangleright \mu_{r=a''})\rangle p \\ \wedge\ a'' = [\![\, q \,]\!]^{a[i]} \\ \wedge\ 1 \leq i \leq n \end{cases}$$

Above, the lookup operator λ takes as parameters three paths p, q, and r and an array of trees a'. When applied to an array of trees $a = [t_1, \cdots, t_n]$, it returns a (i.e., all of its elements, as retuned by the projection π under the first parameter ϵ) where each of its elements has under path p an array of trees obtained from applying the match ($\mu_{r=a''}$) in expression β_i, i.e., following the definition of π, the projection under ϵ is merged with the result of the projection under β_i. For each element $a[i]$ ($1 \leq i \leq n$), β_i matches those trees in a' for which either *i*) there is a path r and the array reached under r equals the array found under $[\![\, q \,]\!]^{a[i]}$ or *ii*) there exist no path r (i.e., its application returns the null array α) and also q does not exist in t_i (i.e., $[\![\, q \,]\!]^{a[i]} = \alpha$).

Example 8. We report the execution of the `lookup` at line 16 of Listing 1.2

$$\alpha \triangleright \lambda_{\texttt{patient\_id=}\alpha'\texttt{.patient\_id}\rangle\texttt{temps}}$$

where α corresponds to the resulting array from the application of the `project` operator at line 15 of Listing 1.2, which has the shape

```
1 α = [ υ {quality:["good"{},"good"{},...], patient_id:["xxx"{}
    ] },
2       υ {quality:["poor"{},"good"{},...], patient_id:["xxx"{}
          ] } ]
```

and where α' corresponds to the array of temperatures that results from the application of the `project` at line 5 of Listing 1.2, as shown at the bottom of Example 5. Then, unfolding the execution of the `lookup`, we obtain the concatenation of the results of two projections, on the only two elements in α. The first corresponds to the projection on ϵ, β_1 while the second corresponds to the projection on ϵ, β_2 where

$$\beta_1 = \beta_2 = \alpha' \triangleright \mu_{\texttt{patient\_id=["xxx" \{\}]}}\rangle\texttt{temps}$$

Below, sub-node `temps` contains the whole array α', since all its elements match `patient_id`.

```
1 [π_{ϵ,β1}(α[1])::π_{ϵ,β2}(α[2])]
2 = [υ {quality:["good"{},"good"{},...], patient_id:["xxx"{}],
3     temps: [
4        υ {temperatures:[36{},...], patient_id:["xxx"{}] },
5        υ {temperatures:[37{},...], patient_id:["xxx"{}] },
6        υ {temperatures:[36{},...], patient_id:["xxx"{}] } ] }
           ,
7  υ {quality:["poor"{},"good"{},...], patient_id:["xxx"{}],
8     temps: [
9        υ {temperatures:[36{},...], patient_id:["xxx"{}] },
10       υ {temperatures:[37{},...], patient_id:["xxx"{}] },
11       υ {temperatures:[36{},...], patient_id:["xxx"{}] } ] }
           ]
```

4 Related Work and Conclusion

In this paper, we focus on ephemeral data handling and contrast DBMS-based solutions wrt to integrated query engines within a given application memory. We indicate issues that make unfit DBMS-based solutions in ephemeral data-handling scenarios and propose a formal model, called TQuery, to express document-based queries over common (JSON, XML, ...), tree-shaped data structures.

TQuery instantiates MQuery [24], a sound variant of the Aggregation Framework [25] used in MongoDB, one of the main NoSQL DBMSes for document-oriented queries.

We implemented TQuery in Jolie, a language to program native microservices, the building blocks of modern systems where ephemeral data handling scenarios are becoming more and more common, like in Internet-of-Things, eHealth, and Edge Computing architectures. Jolie offers variety-by-construction, i.e., the language runtime automatically and efficiently handles data conversion, and all Jolie variables are trees. These factors allowed us to separate input/output data-formats from the data-handling logic, hence providing programmers with a single, consistent interface to use TQuery on any data-format supported by Jolie.

In our treatment, we presented a non-trivial use case from eHealth, which provide a concrete evaluation of both TQuery and MQuery, while also serving as a running example to illustrate the behaviour of the TQuery operators.

Regarding related work, we focus on NoSQL systems, which either target documents, key/value, and graphs. The NoSQL systems closest to ours are the MongoDB [30] Aggregation Framework, and the CouchDB [31] query language which handle JSON-like documents using the JavaScript language and REST APIs. ArangoDB [32] is a native multi-model engine for nested structures that come with its own query language, namely ArangoDB Query Language. Redis [33] is an in-memory multi-data-structure store system, that supports string, hashes, lists, and sets, however it lacks support for tree-shaped data. We conclude the list of external DB solutions with Google Big Table [34] and Apache HBase [35] that are NoSQL DB engines used in big data scenarios, addressing scalability issues, and thus specifically tailored for distributed computing. As argued in the introduction, all these systems are application-external query execution engine and therefore unfit for ephemeral data-handling scenarios.

There are solutions that integrate linguistic abstractions to query data within the memory of an application. One category is represented by Object-relation Mapping (ORM) frameworks [36]. However, ORMs rely on some DBMS, as they map objects used in the application to entities in the DBMS for persistence. Similarly, Opaleye [37] is a Haskell library providing a DSL generating PostgreSQL. Thus, while being integrated within the application programming tools and executing in-memory, in ephemeral data-handling scenarios, ORMs are affected by the same issues of DBMS systems. Another solution is LevelDB [38], which provides both a on-disk and in-memory storage library for C++, Python, and Javascript, inspired by Big Table and developed by Google, however it is limited to key-value data structures and does not support natively tree-shaped data. As cited in the introduction, a solution close to ours is LINQ [16], which provides query operators targeting both SQL tables and XML nested structures with *.NET* query operators. Similarly, CQEngine [39] provides a library for querying Java collections with SQL-like operators. Both solutions do not provide automatic data-format conversion, as our implementation of TQuery in Jolie.

We are currently empirically evaluating the performance of our implementation of TQuery in application scenarios with ephemeral data handling (Internet-of-Things, eHealth, Edge Computing). The next step would be to use those scenarios to conduct a study comparing our solution wrt other proposals among both DBMS and in-memory engines, evaluating their impact on performance

and the development process. Finally, on the one hand, we can support new data formats in Jolie, which makes them automatically available to our TQuery implementation. On the other hand, expanding the set of available operators in TQuery would allow programmers to express more complex queries over any data format supported by Jolie.

References

1. P. R. Orszag, "Evidence on the costs and benefits of health information technology," in *testimony before Congress*, vol. 24, 2008.
2. S. B. Baker, W. Xiang, and I. Atkinson, "Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities," *IEEE Access*, vol. 5, pp. 26521–26544, 2017.
3. W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
4. D. P. Mehta and S. Sahni, *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004.
5. O. Tene and J. Polonetsky, "Big data for all: Privacy and user control in the age of analytics," *Nw. J. Tech. & Intell. Prop.*, vol. 11, p. xxvii, 2012.
6. E. Shein, "Ephemeral data," *Comm. of the ACM*, vol. 56, no. 9, pp. 20–22, 2013.
7. E. Tittel, "The dangers of dark data and how to minimize your exposure." https://tinyurl.com/yb5lxc46, 2014.
8. M. Mostert, A. L. Bredenoord, M. C. I. H. Biesaart, and J. J. M. van Delden, "Big data in medical research and eu data protection law: challenges to the consent or anonymise approach," *European Journal Of Human Genetics*, vol. 24, p. 956 EP, 2015.
9. J. Cheney, S. Lindley, and P. Wadler, "A practical theory of language-integrated query," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 403–416, 2013.
10. L. Welling and L. Thomson, *PHP and MySQL Web development*. Sams Publishing, 2003.
11. MongoDB Inc., "MongoDB Website." https://www.mongodb.com/, 2018.
12. Apache Software Foundation, "CouchDB Website." https://couchdb.apache.org/, 2018.
13. M. Jang, *Linux Annoyances for Geeks: Getting the Most Flexible System in the World Just the Way You Want It.* " O'Reilly Media, Inc.", 2006.
14. Brian Krebs, "Extortionists Wipe Thousands of Databases, Victims Who Pay Up Get Stiffed." https://tinyurl.com/zt53ult, 2017.
15. S. Visveswaran, "Dive into connection pooling with j2ee," *reprinted from JavaWorld*, vol. 7, 2000.
16. E. Meijer, B. Beckman, and G. Bierman, "Linq: reconciling object, relations and xml in the. net framework," in *SIGMOD*, pp. 706–706, ACM, 2006.
17. Niall Gallagher, "CQEngine - Collection Query Engine." https://github.com/npgall/cqengine, 2018.
18. F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with jolie," in *Web Services Foundations* (A. Bouguettaya, Q. Z. Sheng, and F. Daniel, eds.), pp. 81–107, Springer, 2014.
19. Jolie Developers Team, "Jolie Website." https://www.jolie-lang.org/, 2018.
20. N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *PAUSE*, pp. 195–216, Springer, 2017.

21. M. Gabbrielli, S. Giallorenzo, I. Lanese, and S. P. Zingaro, "A language-based approach for interoperability of iot platforms," in *HICSS*, AIS Electronic Library (AISeL), 2018.
22. C. Webster, "From APIs to Microservices: Workflow Orchestration and Choreography Across Healthcare Organizations." https://tinyurl.com/ya57w7wu, 2016.
23. F. Montesi, "Process-aware web programming with jolie," *Science of Computer Programming*, vol. 130, pp. 69–96, 2016.
24. E. Botoeva, D. Calvanese, B. Cogrel, and G. Xiao, "Expressivity and complexity of mongodb queries," in *ICDT*, pp. 9:1–9:23, Schloss Dagstuhl - LZI, 2018.
25. MongoDB Inc., "MongoDB Aggregation Framework." https://docs.mongodb.com/manual/aggregation/, 2018.
26. F. Vigevano and P. D. Liso, "Chapter 11 - differential diagnosis," in *Acute Encephalopathy and Encephalitis in Infancy and Its Related Disorders* (H. Yamanouchi, S. L. Moshé, and A. Okumura, eds.), pp. 81 – 85, Elsevier, 2018.
27. N. Rose, "The human brain project: Social and ethical challenges," *Neuron*, vol. 82, no. 6, pp. 1212 – 1215, 2014.
28. J. A. Bunn, J. W. Navalta, C. J. Fountaine, and J. D. REECE, "Current state of commercial wearable technology in physical activity monitoring 2015–2017," *International journal of exercise science*, vol. 11, no. 7, p. 503, 2018.
29. S. M. Thurman, N. Wasylyshyn, H. Roy, G. Lieberman, J. O. Garcia, A. Asturias, G. N. Okafor, J. C. Elliott, B. Giesbrecht, S. T. Grafton, S. C. Mednick, and J. M. Vettel, "Individual differences in compliance and agreement for sleep logs and wrist actigraphy: A longitudinal study of naturalistic sleep in healthy adults," *PLOS ONE*, vol. 13, pp. 1–23, 01 2018.
30. M. Inc., "Mongodb." https://www.mongodb.com, 2007.
31. A. CouchDB, "Couchdb." https://couchdb.apache.org, 2005.
32. ArangoDB, "Arangodb." https://www.arangodb.com, 2014.
33. S. Sanfilippo and P. Noordhuis, "Redis." https://redis.io/, 2018.
34. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *TOCS*, vol. 26, no. 2, p. 4, 2008.
35. L. George, *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.", 2011.
36. M. Fussel, "Foundations of object-relational mapping," *ChiMu Corporation*, 1997.
37. T. Ellis, "Opaleye." https://github.com/tomjaguarpaw/haskell-opaleye, 2014.
38. G. Inc., "Leveldb." http://leveldb.org, 2017.
39. N. Gallagher, "Cqengine." https://github.com/npgall/cqengine, 2009.

Ephemeral Data Handling in Microservices

***Abstract*—Modern SOA scenarios add, next to the requirements of variety and velocity, *ephemerality*, i.e., the guarantee that processed data do not persist—due to, e.g., storage (IoT) or regulatory (eHealth, GDPR) constraints. We present ongoing work on TQuery, an in-memory query language for ephemeral data-handling, based on a sound formalisation of the widely-used MongoDB query language. We implement TQuery as a library for Jolie; a language for microservices that supports *variety by construction*.**

I. Introduction

Modern Service-Oriented Architectures (SOA) need to address two requirements: velocity and variety [1]. Velocity concerns managing high throughput and real-time processing of data. Variety means that data is represented in heterogeneous formats, complicating their aggregation, query, and storage. Recently, next to velocity and variety, *ephemerality* [2], [3] gained importance, as more and more scenarios require the processed data not to persist, e.g., due to resource constraints — as in the Internet of Things (IoT) [4] — or regulations — e.g., eHealth [5], GDPR [6].

Using a general-purpose language to program complex data-handling logics is time-consuming and error-prone. Thus, developers often prefer to program the data-handling logics in their applications using a query language, paired with an execution engine [7]. The most common configuration for one such solution is to use an external database as execution engine. In that context, the predominant use of structured data-formats (e.g., XML, JSON) pushed the widespread adoption of database management systems (DBMS) based on the NoSQL paradigm and tree-shaped data-structures [1]. However, when considering ephemeral data-handling, DBMSes hinders velocity, due to *resource bottlenecks* and *persistence-related overheads*. Bottlenecks derive from well-known resource constraints, e.g., database connection pools [8]. Overheads are instead typical of the ephemeral case, where data must be first inserted in the database (consuming time and bandwidth), queried, and finally deleted to ensure ephemerality.

These observations pushed us to formalise in [9] a NoSQL-based, in-memory query language, called TQuery, aimed at minimising resource-dependent bottlenecks and eliminating overheads due to data insertions (there is no DB to populate) or deletions (the data disappears when the process handling it terminates). TQuery is inspired by MQuery [10], a sound variant of the MongoDB Aggregation Framework [11]; the most popular query language for NoSQL data handling. The reason behind the formalisation is twofold: *i*) we abstract away implementation details and reason on the overall semantics of our model, to avoid counter-intuitive query behaviours of the Aggregation Framework, as pointed out in [10]; *ii*) we provide a general reference for implementors, which are not tied to a specific technology.

In this work-in-progress paper, we illustrate our implementation of TQuery as an open-source library[1] for the Service-Oriented programming language Jolie [12], [13]. We deem Jolie a suitable choice because: *i*) Jolie programs are natively microservices [14], i.e., state-of-the-art Service-Oriented Architectures; *ii*) the language is successfully used in contexts typical of ephemeral data-handling (Internet-of-Things [15], eHealth [16]); *iii*) Jolie comes with a runtime environment that automatically translates incoming/outgoing data (XML, JSON, etc.) into the native, tree-shaped data values of the language — Jolie values for variables are always trees — and thus it supports *variety by construction*. In section II we illustrate the usage of our library with a comprehensive use-case taken from the eHealth sector. In section III we report preliminary benchmarks, we position TQuery wrt related work, and we draw future directions of research and development.

II. A Use Case from eHealth

We draw our use case from [17], where the authors delineate a diagnostic algorithm to detect cases of encephalopathy. Using TQuery, we follow the principle "data never leave the hospital", in compliance with the GDPR [18]. While the algorithm described in [17] considers a plethora of clinical tests to signal the presence of the neurological condition, we focus on two early markers for encephalopathy: fever in the last 72 hours and lethargy in the last 48 hours. Those data are collectible by commercially-available smart-watches and smart-phones [19]: body temperature and sleep quality. At lines 1–2 of Listing 1, we report, in a JSON-like format, snippets of two kinds of data structures: at line 1 the biometric data collected from the smart-watch of the patient; at line 2 the sleep logs [20]. Both structures are arrays, marked `[ ]`, containing tree-like elements, marked `{ }`. At line 1, for each `date` we have an array of detected temperatures (`t`) and heart-rates (`hr`). At line 2, to each year (`y`) corresponds an array of monthly (`M`) measures, to a month (`m`), an array of daily (`D`) logs, and to a day (`d`), an array of logs (`L`), each representing a sleep session with its start (`s`), end (`e`) and quality (`q`).

On the data structures above, we define a Jolie microservice, reported at lines 4–14, which describes the handling of the data and the workflow of the diagnostic algorithm, using our implementation of TQuery. The example is detailed enough to let us illustrate all the operators in TQuery: `match`, `unwind`, `project`, `group`, and `lookup`. Note that, while in Listing 1 we hard-code some data (e.g., integers representing dates like `20181128`) for presentation purposes, we would normally use parametrised variables.

In Listing 1, line 4 defines a request to an external service, provided by the **`HospitalIT`** infrastructure. The service offers functionality `getPatientPseudoID` which, given some

[1] Address omitted for blind review.

Listing 1. Snippets of biometric (line 1) and sleep logs (line 2) data structures and TQuery-encoded diagnostic algorithm (lines 3–16).

```
1  [{date:20181129,t:[37,..],hr:[64,..]},{date:20181130,t:[36,..],hr:[66,..]},..]
2  [{y:2018,M:[..,{m:11,D:[{d:29,L:[{s:"21:01",e:"22:12",q:"good"},..]},{d:30,L:[{s:..]},..]},..]},..]
3
4  getPatientPseudoID@HospitalIT( patientData )( pseudoID );
5  credentials
6    |> getMotionAndTemperature@SmartWatch |> match { date == 20181128 || date == 20181129 || date ==20181130}
7    |> project { t in temperatures, pseudoID in patient_id } |> temps;
8  detectFever@HospitalIT( temps )( detectedFever );
9  if( detectedFever )
10   credentials |> getSleepPatterns@SmartPhone |> unwind { M.D.L }
11     |> project{ y in year, M.m in month, M.D.d in day,  M.D.L.q in quality }
12     |> match { year == 2018 && month == 11 && ( day == 29 || day == 30 ) }
13     |> group { quality by day, month, year } |> project { quality, pseudoID in patient_id }
14     |> lookup { patient_id == temps.patient_id in temps } |> detectEncephalopathy@HospitalIT
```

identifying `patientData` (acquired earlier), provides a pseudo-anonymised identifier — needed to treat sensitive health data — saved in variable `pseudoID`.

At lines 5–7 and lines 10–14 we use the Jolie chaining operator `|>` to define a sequence of calls, either to external services, marked by the `@` operator, or to the internal TQuery library. The `|>` operator takes the result of the execution of the expression at its left and passes it as the input of the expression on the right.

At lines 5–7 we use TQuery operators `match` and `project` to extract the recorded temperatures of the patient in the last 3 days/72 hours. At line 5 we evaluate the content of variable `credentials`, which holds the certificates to let the Hospital IT services access the physiological sensors of a given patient. In the program, `credentials` is passed by the chaining operator at line 6 as the input of the external call to functionality `getMotionAndTemperature`. That service call returns the biometric data (Listing 1, line 1) from the **`SmartWatch`** of the patient. While the default syntax of a service call in Jolie is the one with the double pair of parenthesis (e.g., at line 4 of Listing 1), thanks to the chaining operator `|>` we can omit to specify the input of `getMotionAndTemperature` (passed by the `|>`) and its output (the biometric data exemplified at Listing 1, passed to the subsequent `|>`). At line 6 we use the TQuery operator `match` to filter all the entries of the biometric data, keeping only those collected in the last 72 hours/3 days (i.e., since `20181130`). The result of the `match` is passed to the `project` operator at line 7, which removes all nodes but the temperatures, found under `t` and renamed **`in`** `temperatures` (this is required by the interface of functionality `detectFever`, explained below). The `projection` also includes in its result the `pseudoID` of the patient, **`in`** node `patient_id`. We finally store the prepared data in variable `temps` (it will be aggregated with the processed sleep logs, at line 14).

At line 8, we call the external functionality `detectFever` to analyse the temperatures and check if the patient manifested any fever, storing the result in variable `detectedFever`.

After the analysis on the temperatures, **`if`** `detectedFever` is true, we continue testing for lethargy. To do that, at line 10, we follow the same strategy described for lines 5–6 to pass the `credentials` to functionality `getSleepPatterns`, used to collect the sleep logs of the patient from her **`SmartPhone`**. Since the sleep logs are nested under years, months, and days, to filter the logs relative to the last 48 hours/2 days, we first flatten the structure through the `unwind` operator applied on nodes `M.D.L` (end of line 10). For each nested node, separated by the dot (`.`), the `unwind` generates a new data structure for each element in the array reached by that node. Concretely, the array returned by the `unwind` operator contains all the sleep logs in the shape:

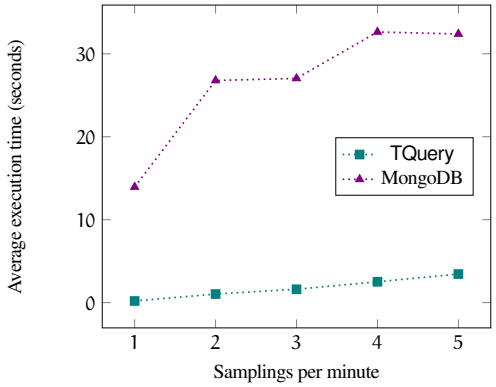
```
[{year:2018,M:[{m:11,D:[{d:29,L:[{s:"21:01",e:"22:12",q:"good"}]}]}]}
 {year:2018,M:[{m:11,D:[{d:29,L:[{s:"22:36",e:"22:58",q:"good"}]}]}]}]
```

where there are as many elements as there are sleep logs and the arrays under `M`, `D`, and `L` contain only one sleep log. Once flattened, at line 11 we modify the data-structure with the `project` operator to simplify the subsequent chained commands: we rename the node `y` **`in`** `year`, we move and rename the node `M.m` **`in`** `month` (bringing it at the same nesting level of `year`); similarly, we move `M.D.d`, renaming it `day`, and we move `M.D.L.q` (the log the quality of the sleep), renaming it `quality` — `M.D.L.s` and `M.D.L.e`, not included in the `project`, are discarded. On the obtained structure, we filter the sleep logs relative to the last 48 hours with the `match` operator at line 12. At line 13 we use the `group` operator to aggregate the `quality` of the sleep sessions recorded in the same day (i.e., grouping them **`by`** `day`, `month`, and `year`) and use the `projection` to keep only the aggregated values of `quality` (getting rid of `day`, `month`, and `year`); we also include under node `patient_id` the `pseudoID` of the patient. That value is used at line 14 to join, with the `lookup` operator, the obtained sleep logs with the previous values of temperatures (`temps`). The resulting, merged data-structure is finally passed to the **`HospitalIT`** services by calling the functionality `detectEncephalopathy`.

III. Benchmarks and Discussion

Benchmarks. As a preliminary result, we benchmarked the query at lines 6–7 of Listing 1, testing our implementation of TQuery against a comparable architecture that uses MongoDB to handle the queries. Namely, we programmed two microservices: `QS` contains the implementation at lines 6–7 of Listing 1; `MS` implements the same logic in terms of MongoDB queries: *i*) we insert the data in the database, *ii*) we send the query (match and project) as one instruction to the database, and *iii*) we delete the inserted data to ensure ephemerality. To run our tests, we use 5 copies of the data-structure at line 1 of Listing 1. All copies cover 365 days of recordings but at increasing sampling rates, i.e., the first has 1440 samplings per day (one per minute) and the other four increase the sampling to 2, 3, 4, and 5. We simulate bursts of requests by sending 4 subsequent batches of requests, each with 10 concurrent requests (amounting to 40 requests in

total). A third microservice loads the data and sends 10 separate requests to `QS` (resp. `MS`) at a time. We draw our benchmarks in the figure below, reporting the average time over the 40 requests, for each sampling. In `QS` we start the timer before executing the first query instruction (`match`) and we stop it after we obtain the result of the last (`project`). In `MS` we start the timer before executing the insertion of the data in the database and stop it after we queried and deleted the data. We run our benchmarks on a machine equipped with a 2.6GHz quad-core Intel Core i7 processor and 16GB RAM. The machine runs macOS 10.14, Java 11, Jolie 1.8-beta, and MongoDB 4.



In all instances, TQuery is faster than the alternative based on MongoDB. Intuitively, this is due to the combination of the overheads of *i*) additional data transmission (in particular insertion of the data and return of the result of the query), *ii*) writings on disk (to ensure data persistence), and *iii*) the concurrent access to the database connection channel.

Discussion. We present ongoing work on and illustrate the usage of our implementation of TQuery as a library for the Jolie Service-Oriented language. Thanks to Jolie, our implementation of TQuery enjoys variety-by-construction, i.e., it provides a consistent interface to query any data-format that is automatically and efficiently converted by the Jolie runtime (JSON, XML, etc.).

Regarding related work, we do not compare with DBMS systems in general, since we rule out their usage in the context of ephemeral data-handling, with the exception of ArangoDB [21]; an in-memory DBMS that can support JSON-like data-structures. The main differences with TQuery are: *i*) there is still overhead due to moving the data in memory between the database and the host program (assuming the in-memory database instance vanishes with the program running it) and *ii*) since ArangoDB supports multiple data-models, it comes with a query language that is not specifically suited for tree-shaped data-structures. Another solution close to ours is LINQ [22], which, similarly to TQuery, is a integrated, in-memory query language, however, similarly to ArangoDB, it provides SQL-like operators, which are not specifically purposed for tree-shaped data-structures.

Besides finalising the implementation of TQuery, we will conduct more extensive studies to evaluate its performance wrt different application contexts. We will also benchmark our implementation against a significant set of alternatives (SQL databases, LINQ, etc.). We also plan to expand the set of operators in TQuery, to simplify the definition of more complex queries.

References

[1] D. P. Mehta and S. Sahni, *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004.

[2] O. Tene and J. Polonetsky, "Big data for all: Privacy and user control in the age of analytics", *Nw. J. Tech. & Intell. Prop.*, 2012.

[3] E. Shein, "Ephemeral data", *Comm. of the ACM*, 2013.

[4] S. B. Baker *et al.*, "Internet of Things for Smart Healthcare: Technologies, Challenges, and Opportunities", *IEEE Access*, 2017.

[5] P. R. Orszag, "Evidence on the costs and benefits of health information technology", in *testimony before Congress*, 2008.

[6] M. Mostert *et al.*, "Big data in medical research and eu data protection law: Challenges to the consent or anonymise approach", *European Journal Of Human Genetics*, 2015.

[7] J. Cheney *et al.*, "A practical theory of language-integrated query", *ACM SIGPLAN Notices*, 2013.

[8] S. Visveswaran, "Dive into connection pooling with j2ee", *reprinted from JavaWorld*, 2000.

[9] Authors omitted for blind review, "Ephemeral data handling in microservices", Tech. Rep., 2019.

[10] E. Botoeva *et al.*, "Expressivity and complexity of mongodb queries", in *ICDT*, Schloss Dagstuhl - LZI, 2018.

[11] MongoDB Inc., *MongoDB Aggregation Framework*, https://docs.mongodb.com/manual/aggregation/, 2018.

[12] F. Montesi *et al.*, "Service-oriented programming with jolie", in *Web Services Foundations*, A. Bouguettaya *et al.*, Eds., Springer, 2014.

[13] Jolie Developers Team, *Jolie Website*, https://www.jolie-lang.org/, 2018.

[14] N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow", in *PAUSE*, Springer, 2017.

[15] M. Gabbrielli *et al.*, "A language-based approach for interoperability of iot platforms", in *HICSS*, AIS Electronic Library (AISeL), 2018.

[16] C. Webster, *From APIs to Microservices: Workflow Orchestration and Choreography Across Healthcare Organizations*, https://tinyurl.com/ya57w7wu, 2016.

[17] F. Vigevano and P. D. Liso, "Chapter 11 - differential diagnosis", in *Acute Encephalopathy and Encephalitis in Infancy and Its Related Disorders*, H. Yamanouchi *et al.*, Eds., Elsevier, 2018.

[18] N. Rose, "The human brain project: Social and ethical challenges", *Neuron*, 2014.

[19] J. A. Bunn *et al.*, "Current state of commercial wearable technology in physical activity monitoring 2015–2017", *International journal of exercise science*, 2018.

[20] S. M. Thurman *et al.*, "Individual differences in compliance and agreement for sleep logs and wrist actigraphy: A longitudinal study of naturalistic sleep in healthy adults", *PLOS ONE*, Jan. 2018.

[21] ArangoDB, *Arangodb*, https://www.arangodb.com, 2014.

[22] E. Meijer *et al.*, "Linq: Reconciling object, relations and xml in the. net framework", in *SIGMOD*, ACM, 2006.