Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

GraphQL Federation: A Model-Based Approach

Patrick Stünkel[a] Ole von Bargen[b] Adrian Rutle[a]
Yngve Lamo[a]

a. Høgskulen på Vestlandet, Bergen, Norway
b. VHV solutions GmbH, Hanover, Germany

Abstract The *Graph Query Language (GraphQL)* is a framework for developing Web Services, which works on the domain model level rather than the functions. While the declarative nature of this framework has already attracted the interest of practitioners in both academia and industry, it still lacks integration capabilities. This shortcoming has been recognized in the industry; there exists a first tool creating a federation—a virtual integrated system—on top of instrumented systems. Being relatively new, it suffers from a few practical and conceptual shortcomings, such as consolidation of (conflicting) schemas and identification of multiple types. In this paper, we will analyze the federation challenge and propose a solution utilizing multi-view modeling and domain specific modeling. Our approach is accompanied by a proof-of-concept implementation and provides a model-based presentation of the GraphQL framework.

Keywords GraphQL; Schema Definition Language; System Integration; Web Services; Federated Systems; Model Merging; Correspondence Links

1 Introduction

Microservices are a current industry trend, which encourages the design of software systems as a composition of small connected components [FML17, AAE16, FL14]. The main advantage of this approach is *scalability*, both in terms of personnel organization and resource distribution, which is fueled by other trends such as build automation, application containers and Continuous Integration (CI)/Continuous Deployment (CD) [AL19, Ric15]. However, all of this comes at the cost of higher *complexity*: The necessity of relying on network communication, possibly being unreliable, slow and insecure [TS07], forces developers to implement counter-measures against message loss, timeouts and intrusion.

Consequently, there is a rise of tools and frameworks to leverage the development of distributed software systems. A notable representative is *GraphQL*, an open-source framework for developing *Web Services (WS)*. It receives increasing interests by both

Patrick Stünkel, Ole von Bargen, Adrian Rutle, Yngve Lamo. *GraphQL Federation: A Model-Based Approach*. In Journal of Object Technology, vol. 19, no. 2, 2020, pages 18:1–21. doi:10.5381/jot.2020.19.2.a18

industry and academia, see e.g. [HP18, UKT+19, MRKIP+19]. The idea behind GraphQL is that the server should not provide its interface as a collection of functions, as it is practice in the well-known WS-protocols SOAP and REST, but instead provide its domain model as a *graph*. The benefit is that a client can flexibly specify *what* part of the domain model graph it is interested in rather than relying on the existence of a function which provides access to this information. This enables server-side and client-side teams to work mostly independently, after having agreed on the common domain model. GraphQL can be considered as an instance of the Model Based Software Engineering (MBSE) paradigm [BCW17]. The domain model graph is described by a so-called *Schema Description Language (SDL)* which is the foundation for the query language available to the clients. The queries are formulated declaratively and the technical details of communication and encoding are handled by the framework.

While GraphQL facilitates the development of a WS-interface for a *single* application, it lacks means for integrating multiple interfaces, an important requirement for distributed systems. One example use case is the retrieval of information, which is scattered over multiple local services [DHI12]. Without a proper integration functionality, the client itself has the obligation to send requests to all relevant services and consolidate the responses correctly.

As a solution to this issue, we develop a framework for creating a *federation* (common interface) on top of existing GraphQL WS-interfaces. Our solution utilizes a *Domain Specific Language (DSL)* for describing schema *correspondences*—also called *traceability links*—which are a common means for relating multiple modeling artifacts [SKLR18, FKWVH19, ARNRSG06]. The framework is non-intrusive and technology independent, and it facilitates the consolidation of multiple (conflicting) schemas where multiple type definitions are flexibly be related or identified. In addition, we compare our solution and present its advantages over an existing tool called *Apollo Federation* [Met]. Furthermore, our investigations are accompanied by an MBSE-oriented presentation of the GraphQL framework and a proof-of-concept implementation of our solution.

Outline. In following section, we will both present GraphQL (2.1) as an example of MBSE and state the integration problem (2.2) we are aiming to solve. Section 3 provides a short overview on MBSE literature dealing with similar problems. Section 4 presents concrete solutions to the problem. First, the tool Apollo Federation is analyzed in section 4.1 and afterwards we present our solution in Sect. 4.2. Both approaches are finally evaluated in section 5 before we finally wrap up our investigations and provide an outlook on future work in section 6.

2 Background

There is no fixed definition of the term *Microservice*, however, there is a commonly used definition attempt as an "approach of developing a single application as a suite of small services" [FL14]. It is closely related to *Service-Oriented Architecture (SOA)* [Jos07] since it shares the same underlying idea. The major difference is that SOA focuses on integration and service orchestration (usually requiring a sophisticated middleware), whereas Microservices focus on separating the business logic into small independent components [CDT18, FML17, Ric15]. SOA and Microservices imply a distributed system architecture, which adds further complexity in terms of communication, synchronization, safety and security [TS07].

```
type Query {
  customer(customer: ID!): Customer
  allCustomers: [Customer]
  store(store: ID!): Store
  allStores: [Store]
}
type Mutation {
  createCustomer(name: String!, email: String): Customer!
  updateCustomer(customer: ID!, name: String, email: String): Customer
  deleteCustomer(customer: ID!): Customer
  setAddress(customer: ID!, street: String, city: String, postalCode: String, state: String,
             country: String): Customer
  createPurchase(customer: ID!, date: String!, store: ID!): Purchase
  addItemToPurchase(purchase: ID!, product: ID!, quantity: Int): PurchaseItem
}
```

(a) Facade Part

```
type Customer {
  id: ID!
  name: String!
  email: String
  address: Address
  purchases: [Purchase]
}

type Purchase {
  id: ID!
  date: String!
  customer: Customer!
  store: Store!
  items: [PurchaseItem]
}
```

(b) Types Part 1

```
type PurchaseItem {
  productID: ID!
  quantity: Int
}
type Store {
  id: ID!
  manager: ID!
  purchases: [Purchase]
  location: Address!
}
type Address {
  street: String
  city: String
  postalCode: String
  state: String
  country: String
}
```

(c) Types Part 2

Figure 1 – Sales System GraphQL Schema

2.1 GraphQL

Traditionally there are two WS-protocols: SOAP [Wor07] and REST [FT00]. The SOAP protocol requires to explicitly specify all accessible methods including input and output data structures in a schema, whereas REST uses the information encoded in the URL and HTTP-method to describe the service interface. The main disadvantage of SOAP is seen in its *complexity* and the overhead imposed through the explicit schema. REST on the other hand is more lightweight but relies on developer *discipline*, i.e. that interfaces are designed and used in accordance to the REST guidelines and that input and output data structures are well documented. GraphQL [Fac18] steps in as yet another alternative for developing WS, which seeks to combine advantages of both approaches. It was originally developed by Facebook and eventually open sourced in 2015. Since then, it is been adopted by other companies such as Twitter, Airbnb, and Github, also there are first use cases in academia, see e.g. [UKT+19, MRKIP+19]. The name *GraphQL* arises from the fact that every server interface has to define its underlying domain model in a schema representing a graph. This definition is given in a textual *Schema Definition Language (SDL)*. Figure 1 exemplifies a GraphQL Schema for a simplified *sales* application of a hypothetical retail company.

The keyword `type` initiates the declaration of a so-called Object Type (think *class* in UML, and see Figure 3). An Object Type has a name and comprises multiple Fields, which again have a name and a return-Type. The return-Type can either be a Scalar

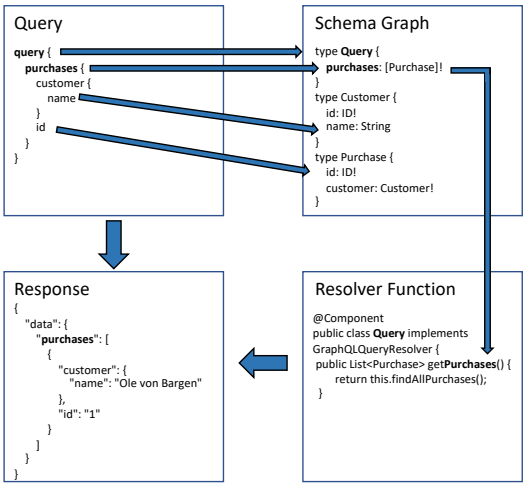

Figure 2 – GraphQL Query Mechanism explained

Type (think *data type* in UML) such as `String`, `Int` or another Object Type. Fields can be marked as *mandatory* (denoted by an exclamation mark `!`) or *list-valued* (denoted by brackets `[ ]`), i.e. per default a Field is *optional* and is thus allowed to return a `null`-value. Furthermore a Field may have a list of Arguments. An Argument has a name and a type, which has to be a Scalar Type. The purpose of an argument is to be passed to the underlying Resolver-mechanism to perform *filtering*.

Every Schema may contain the special entities `Query` and/or `Mutation`; see Figure 1a. These types are intended to serve as an *entry point* or *facade* [GHJV95]. They are *singleton* Object Types [GHJV95], whose Fields represent methods for retrieval and modification of data. Fields of a `Query` are intended to be side-effect-free read-only functions, whereas Fields of a `Mutation` are methods changing the underlying data set. Every Schema has to be accompanied by a set of Resolver-functions, one for each Field. Provided with the parent object and values for all Field Arguments, a Resolver returns instances of the Return-Type, i.e. objects or scalar values. This allows developers to perform intermediate business-specific operations and to use any data store of their choice. A Schema together with a set of Resolvers constitutes a GraphQL Endpoint.

A client interacts with an Endpoint by sending a Query to it. The Query is formulated in the eponymous GraphQL. A graph Query is basically a tree of Field names from the respective Schema. The root of the tree has to either be a reference to the `Query` type or the `Mutation` type. A Query is evaluated by the GraphQL framework, which is illustrated in Figure 2. Hierarchically, the Resolvers are called, providing (lists of) objects or values, which are assembled into a single Response JSON Document.

GraphQL can be seen as an example of MBSE since it encourages a declarative approach focusing on domain models rather than technical details. The SDL is a *Domain Specific Language* [DK04], whose metamodel is depicted in the highlighted fragment in Figure 3, and a schema is a model conforming to this metamodel. Semantics for a schema is given by the collection of all data sets conforming to the schema, and one particular data set is provided by a concrete resolver implementation. Putting this into the metamodeling hierarchy [Kü06], the GraphQL metamodel in Figure 3 is an M2-model, every GraphQL schema is an M1-model, and data sets behind these schemas are M0-models.

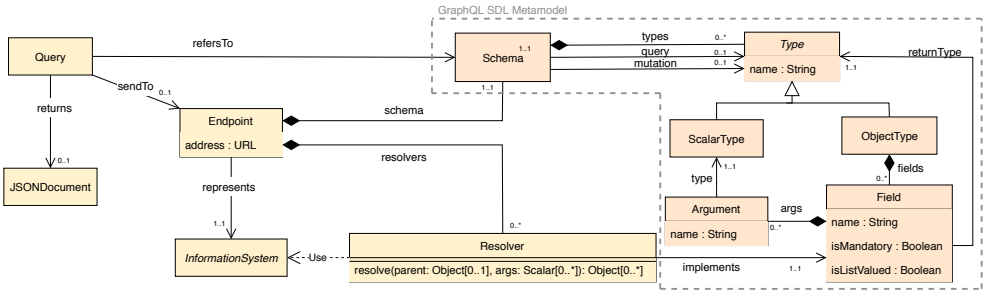

Figure 3 – Metamodel of GraphQL concepts

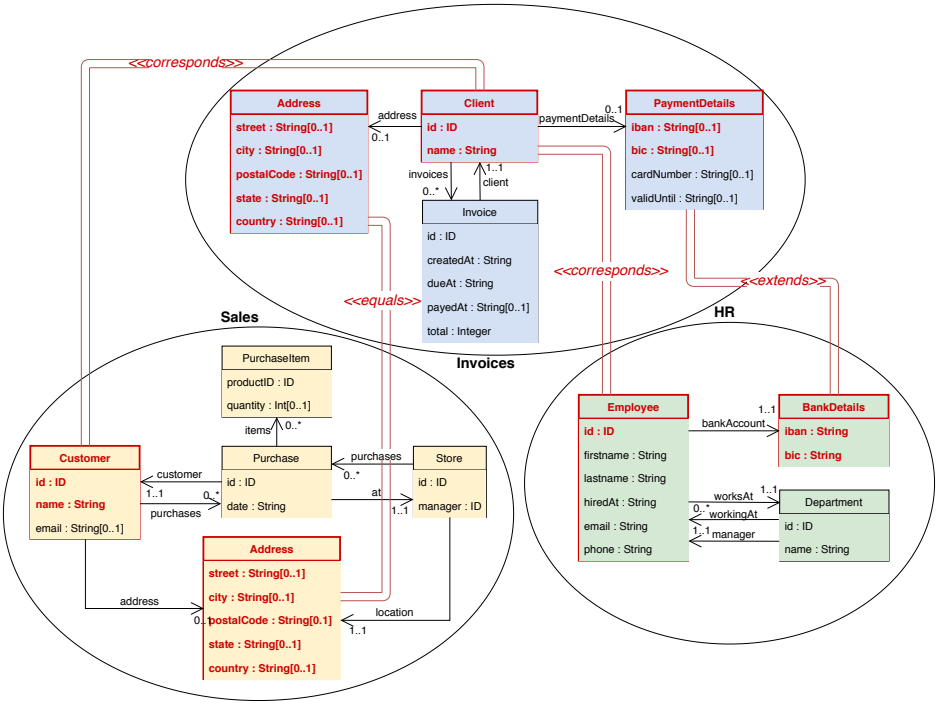

Figure 4 – Domain Models of the Systems

Returning to our retail company example, for the remainder of this paper let us assume that the company has adopted the current Microservice trend and organizes its system infrastructure into cross-functional teams maintaining small autonomous components. Hence, in addition to the sales system endpoint EP_1, shown in Figure 1, there is another system EP_2 for writing *invoices*, and finally the company bought in a *HR* system EP_3. Using the metamodel hierachy described above, we can denote the schemas of these three endpoints in a compact and familiar UML/MOF-oriented notation in figure 4. Ovals and different shading indicate the systems' boundaries (ignore the red color for the time being). Every *class* becomes an Object Type, every *attribute* becomes a Field returning a Scalar Type, and every directed reference becomes a Field returning an Object Type. The multiplicity `1..1` indicates the respective field being mandatory (`!`), and `0..*` indicates the field being list-valued (`[ ]`). Finally, let us assume there are `Query` and `Mutation` types (not shown) providing the regular *Create, Read, Update, and Delete (CRUD)* methods.

2.2 Problem Statement: Federation

In SOAs, systems are integrated on the service level via sophisticated orchestration mechanisms. This generally requires a *canonical data model* [CDT18]. In a Microservice architecture, which heavily encourages decoupling, such canonical data models are uncommon and unfeasible, see e.g. figure 4, where the three domain models have been developed independently. This leads to data integration issues.

In the running example, our Retail company now faces several challenges: First, there is a growing problem of *redundant* data and thus possibly inconsistent data, e.g. in the past there had been issues where customers relocated to a new address but the records had not been updated consistently such that invoices were not delivered to the right customers. Secondly, there is a difficulty in implementing *new use cases*, e.g. the company wants to implement a policy, which gives their employees discounts on purchases in the company's stores. To implement this use case it is necessary to align the data set of the *HR* system with the two other systems. Thirdly, the company's CEO is interested in the *reporting and analysis* of the data contained in the information systems. These three issues are examples of the most common reasons for data integration, for a detailed treatment we refer to [DHI12].

Integration can be defined as a process that turns separate systems into a single (possibly distributed) system. The systems we are considering in the example are GraphQL endpoints where an endpoint is given by a schema and a set of resolvers. Thus, endpoint integration means to consolidate the schemas of EP_1, EP_2 and EP_3 and to merge their data sets. In the database and information systems domain, integration often appears under the term *federation*. A federated system is a lightweight or even virtual abstraction over the integrated local system, which remain autonomous. A federated endpoint addresses the practical issues mentioned above: A unified data set allows a detection of address duplicates, to identity persons who are both customers and employees, and to have a global reporting. The goal is illustrated in Figure 5. There are the three information systems *Sales*, *Invoice*, *HR*, connected by a network, and accessible via the GraphQL endpoints EP_1, EP_2 and EP_3. The federated endpoint is visualized by the dotted components and referred to as *Backoffice*. It is visible to internal and external clients through the endpoint EP^+, and behaves as a regular GraphQL endpoint.

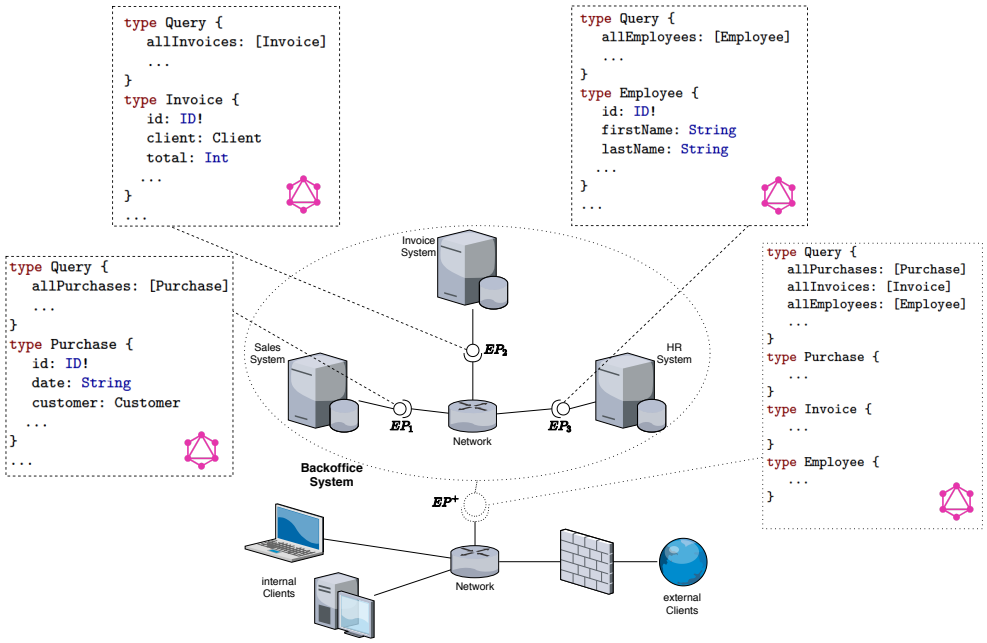

Figure 5 – GraphQL Federation: an overview

3 State of the Art

Federation can be considered as an instance of *multi-view(-point) modeling*: The development of a complex system usually requires to work with multiple (potentially overlapping) *views* of the system. It is a long standing and active research domain—see [FKN+92] for one of the first mentions and [CCP19, BBCW19] for very recent surveys on the topic—and it is further related to important research domains such as *megamodeling* [Ste20], *inter-model consistency* [KM18], *model synchronization* [AC08], and *bidirectional transformations (bx)* [CFH+09]. In [ATM15], the authors distinguish between *projective* and *synthetic* multi-view modeling. The projective approach assumes a pre-existing all-encompassing *system model*. Views are derived by a projection on certain parts of the system model. Two views share elements when they refer to the same elements in the system model. In the *synthetic* approach, there is no pre-existing system model and thus the system must be synthesized by view integration. *Federation* is evidently an instance of synthetic multi-view modeling and the literature reports on different view integration approaches presented in Fig. 6, see [KM18, CCP19, dLGKH18] for further references.

One of the most widespread approaches is *model merging* [BCE+06], i.e. to create our own system model as new artifact, which encompasses all views and for every view there exists an embedding into it. A different approach is *cross-referencing* where external links are added on top of the views establishing cross-references between elements from different views, see e.g. [KG19, FKWVH19]. The *model extension* approach [KM18] does not rely on *global* notions such as a merging or cross-references but instead augments the views *locally* with elements from other views. *Heterogeneous transformations* are very common in model synchronization scenarios and bx [CFH+09]. which considers model transformations for every pair of views to allow translations

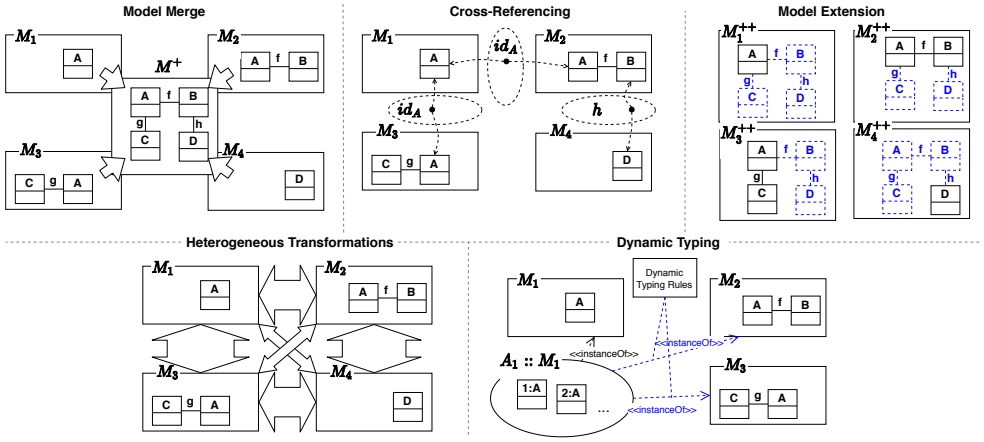

Figure 6 – Model Integration approaches

between them. Another recent and promising approach relies on concepts from multi-level modeling [AK01, Kü06, MGdL17, MWR+19] by considering the typing relation between models and their metamodels as *dynamic* and *non-binary*: Elements in a view model may on the one hand change their type over time, and on the other hand may be typed over elements from multiple metamodels at the same time. Typical examples are *roles* [KBGA15] and *facets* [dLGKH18]. This removes the necessity to perform further integration on the metamodel level, however, there is commonly a need for typing rules to control correct typing. Such typing rules share similarities with the cross-references mentioned above.

4 Solutions

In this sections, we investigate how endpoint federation may be realized. First, we analyze an existing approach called *Apollo Federation* before we propose our solution, which aims to leverage some shortcomings of the former.

4.1 Existing Solution: Apollo Federation

The most popular *JavaScript* implementation of GraphQL is *Apollo GraphQL*[1], which is developed by the Meteor Development Group. They realized that it gets increasingly inefficient and difficult to force a whole organization to expose all its services behind a single GraphQL Endpoint. Thus, they developed the tool *Apollo Federation* [Met] on top of GraphQL, which enables organizations to split the definition of a single endpoint (Schema and Resolvers) into smaller definitions and develop them independently of each other, which enables *Separation of Concerns (SoC)*.

First and foremost, Apollo Federation provides a *language extension* of the GraphQL SDL. The new language features are a new keyword (`extend`) and the directives `@key` and `@external`. Directives are the GraphQL equivalent of annotations; i.e., they have no predefined meaning but can be used by a concrete GraphQL implementation to introduce proprietary features. The most fundamental feature is the keyword `extend`,

[1] `https://www.apollographql.com/`

```
extend type Query {
    clients: [Client]
    ...
}
type Partner
@key(fields: "id")
@key(fields: "name") {
    id: ID!
    name: String
    address: Address
    invoices: [Invoice]
}
...
```

(a) Schema Extension *Invoices*

```
extend type Query {
   customers: [Partner]!
   ...
}
extend type Partner @key(fields:"id") {
   id: ID!
   name: String @external()
   address: Address @externa()
   email: String
   purchases: [Purchase]
}
...
```

(b) Schema Extension for *Sales*

Figure 7 – Appollo Federation Syntax Extension Examples

which can be put in front of a type definition, meaning that this type now represents an extension of an existing type in another schema. To see how this concept can be used to identify the two `Customer` types in *Sales* and `Client` in *Invoices* (compare Figure 4) have a look at Figure 7: *Invoices* provides the initial definition and *Sales* extends it and because type extensions are identified by name we have to give a common name to `Customer` and `Client`, `Partner` in this case. In the *Apollo Federation* parlance, `Partner` is a called an *entity*. Instances of entities can be globally identified such that the federated endpoint can consolidate data from multiple endpoints via identifying elements with the same key. Keys are defined using the `@key` directive on the initial type definition. A `@key` refers to at least one of the entity's fields meaning that two entity instances are identified if the values to all these field are identical (AND-semantics). There can be multiple `@key` directives, which means that they serve as *alternative* (OR-semantics) keys. A type extension has to pick one of the declared keys from the original definition, see Figure 7b. Finally, an extended type can introduce new fields, e.g. `email` and `purchases` for which regular resolvers have to be implemented.

This extended SDL has ramifications on the implementation of the respective resolvers. First, for every entity type definition there has to be a special resolver retrieving an entity instance for a given key. Resolvers for a type extension must only provide resolvers for every key attribute and the non-external fields. Upon querying an endpoint with a type extension, values for the `@external` fields are retrieved by asking the endpoint containing the original definition, which is possible due to the global identification mechanism. After, having augmented the endpoints with the Apollo Federation features, the endpoints can automatically be integrated into a federated endpoint, which aggregates the type definitions from all local endpoints. For our example, this federated endpoint comprising *Sales* and *Invoices* will contain a single `Address` type, a single `Partner` comprising the fields `id`, `name`, `email`, `address`, `invoices`, and `purchases`, and there will be a single `Query` type aggregating the query methods from all endpoints.

Thus, Apollo Federation facilitates the Federation problem described in Sect. 2.2. However, it is important to note that Apollo Federations architecture is actually based on projective multi-view model management. It assumes an unambiguous underlying system model that is separated into multiple schemas connected by type extensions (established via equality of type names). The Apollo Federation tool works well for scenarios where the programmer has access to every endpoint implementation. But this is not always the case especially when there is bought-in software, such as e.g.

the HR system in our example. Thus, Apollo Federation has no way of reconciling conflicting views.. This and other limitations of Apollo Federation are discussed in the list below:

Lim#1 The requirement for an **Underlying System Model** requirement may pose problems in certain system integration scenarios where one cannot modify the existing endpoints: There is no way to reconcile "conflicting" views, e.g. not having declared `name` and `address` in Figure 7 as `@external` would have resulted in an error.

Lim#2 Type Extension is a **binary relation**. This has implications on the amount of data that can be aggregated within a single query. A query against a field, which returns a type extension will aggregate data from at most two endpoints. Considering the running example, when we include the *HR* system into the federation and treat `Employee` as a type extension of `Customer` as well, we can retrieve all `Partner`s, which are `Customer`s and `Client`s or `Customer`s and `Employee`s but not altogether.

Lim#3 **Identification Capabilities** are **limited**; Apollo Federation allows to identify elements by alternative and composite keys but it is limited to field-references only. There are no possibilities to perform intermediate computations. For example, if we want to integrate the data set of the *HR* system, it is not possible to state that the value of `name` for a `Client/Customer` is equal to the concatenation of `firstname` and `lastname` of an `Employee`.

Lim#4 Apollo Federation is **intrusive**. By using the *model extension* approach for view integration, compare Sect. 3, each endpoint is augmented with global information (additional schema language features and special resolver functions). Thus, the local endpoints are not unaware that they are participating in a federated system.

Lim#5 Apollo Federation is **technology dependent** by requiring every endpoint, which participates in a federation, to follow and implement a specific protocol. Thus, it currently only works with a chosen selection of GraphQL implementations, see [Met]

4.2 Our Solution: Schema Correspondences

Our goal in this paper is to provide a solution for endpoint federation, which addresses the limitations mentioned above. Thus, we want to develop a non-intrusive and technology independent framework, which allows to consolidate multiple possibly conflicting schemas where more than two type definitions can be related and elements can flexibly be identified.

To fulfill non-intrusiveness (see Lim#4) we decide to use a *cross-referencing approach*, see Sect. 3, i.e. we add structural cross-reference links on top of the existing endpoint schemas without changing them. These links will be called *correspondences* [SKLR18] and establishes a relation between types or attributes from different schemas that refer to the same real-world concept.

In Figure 4 we highlighted the correspondences in our example in red. The figure illustrates how correspondences accommodate for different types of cross-references, e.g. the same type `Address` appears in *Sales* and *Invoices*, the type `PaymentDetails` in *Invoices* extends the type `BankDetails` in *HR*, and there is the structural correspondence between `Customer`, `Client` and `Employee`, discussed above. Note, that we

```
1  endpoints {
2     "http://localhost:4011/" as sales
3     "http://localhost:4012/" as invoices
4     "http://localhost:4013/" as hr
5  }
6  correspondences {
7     relate {sales.Query, invoices.Query, hr.Query} as Query with {
8        relate { sales.Query.customers, invoices.Query.clients, hr.Quey.employees} as partners
9     }
10    relate { sales.Customer, invoices.Client, hr.Employee }
11       as Partner
12       with {
13          relate { sales.Customer.id, invoices.Client.id, hr.Employee.id } as id
14          relate { sales.Customer.name, invoices.Client.name } as fullName
15          relate { sales.Customer.address, invoices.Client.address } as address
16          relate { invoices.Client.paymentDetails, hr.Employee.bankAccount } as financialDetails
17          identify where {
18             or {
19                equals { sales.Customer.id, invoices.Client.id }
20                equals {
21                   invoices.client.Name,
22                   concat {hr.Employee.firstname," ",hr.Employee.lastname}
23                }
24             }
25          }
26       }
27    relate { invoices.PaymentDetails, hr.BankDetails } as FinancialDetails
28       with {
29          relate { invoices.PaymentDetails.iban, hr.BankDetails.iban } as iban
30          relate { invoices.PaymentDetails.bic, hr.BankDetails.bic } as bic
31          identify where {
32            and {
33                equals { invoices.PaymentDetails.iban, hr.BankDetails.iban }
34                equals { invoices.PaymentDetails.bic, hr.BankDetails.bic }
35            }
36          }
37       }
38    relate { sales.Address, invoices.Address } as Address
39       with {
40          relate all
41          identify where all equals
42       }
43 }
```

Figure 8 – Correspondence DSL example

do not require the existence of systems model (Lim#1) and are able to deal with conflicting views. Every type and field is considered separate if not otherwise related by a correspondence.

Hence, we have to develop a DSL to allow for correspondence definition. To fulfill the requirements of the federation problem, this DSL must offer two features: On the one hand it must allow the definition of multi-ary (see Lim#2) type and field correspondences. On the other hand it must allow the definitions of data identification mechanisms. Apollo Federation uses keys for this and we also adopt this idea but allow flexible expressions that can be used for keys (see Lim#3)

Figure 8 presents a first proposal of what such a language might look like. It encodes the correspondences of our example from Figure 4. First, the local endpoints participating in the federation have to be specified. They are identified via their URL and for the remainder identified by a symbolic name (lines 2-4). Afterwards, type correspondences (initiated by the keyword `relate`) are given. We establish correspondences among the three `Query` root types (lines 7-9), among `Client`, `Customer` and `Employee`, (lines 10-26), among `PaymentDetails` and `BankDetails` (lines 27-37), and

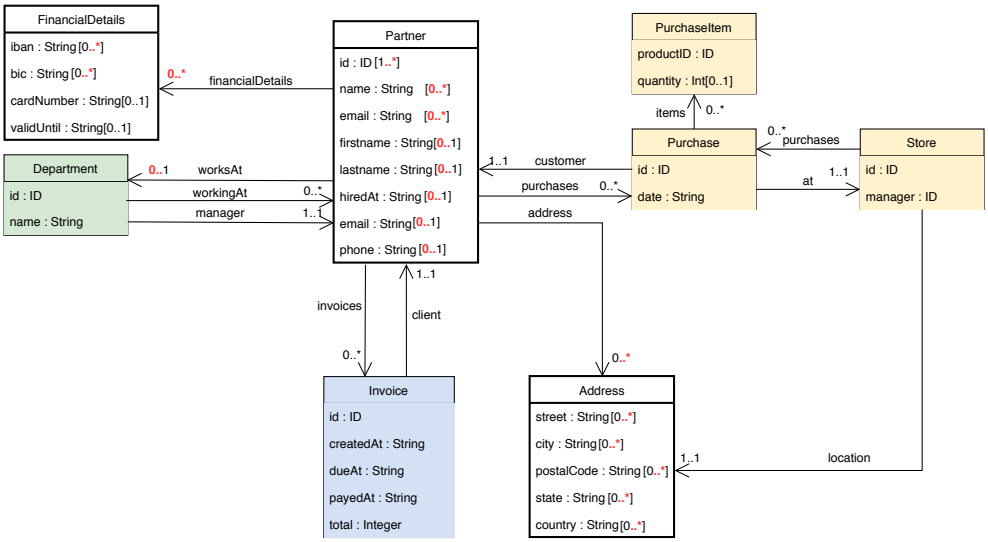

Figure 9 – Merged Domain Models for the Federation

for the shared `Address` type (lines 38-42). A type correspondence further contains field correspondences, which are initiated by the keyword `with` and follow a similar structure as type correspondences. Also, a type correspondence may introduce an identification mechanism (lines 17-25, 31-36, and 41). An identification is a boolean expression in disjunctive normal form, with literals being equalities between field references or scalar expressions built over field references. The idea is that a disjunction translates to alternative keys and a conjunction translates to a composite key. The ability to use arbitrary expression addresses Lim#3. E.g. we assume that `Clients` (*Invoices*) can be identified with `Customers` (*Sales*) via their `id` and `Clients` can be matched with `Employees` (*HR*) by comparing their `name` with concatenation of `first-` and `lastname`. Finally we introduce some syntactic sugar (line 40-41) to identify types, which are structurally identical.

Eventually, our tool has to produce a running endpoint, i.e. a global schema and resolvers for every field in that schema. The correspondence definition in Figure 8 is structurally different from a GraphQL schema, however, luckily there are extensive results on automatic model merging using global correspondences, see e.g. [SE05, BCE+06, RC13]. We adopt the algorithm that is formally described in [SKLR18]. The respective theoretical construction is called a *pushout*, which is intuitively described as taking the *union* of all schemas wherein corresponding elements are *identified*. The resulting merge of the domain models (schemas) for our running example is shown in figure 9. Note that the merge construction changes multiplicities for fields that are identified (highlighted in red) because a merged type may contain more or less data than its local equivalent.

Next, we have to implement resolvers s for our new endpoint. Since the federated endpoint must not suddenly "invent" new information, these resolvers are implemented via delegation to existing resolvers in other endpoints. To discuss this idea consider Figure 10, which shows a metamodel of our framework.

Every definition like the one in Figure 8, we call it CorrSpec, refers to at least to existing endpoints and contains correspondences. An endpoint URL extends to a `Schema`

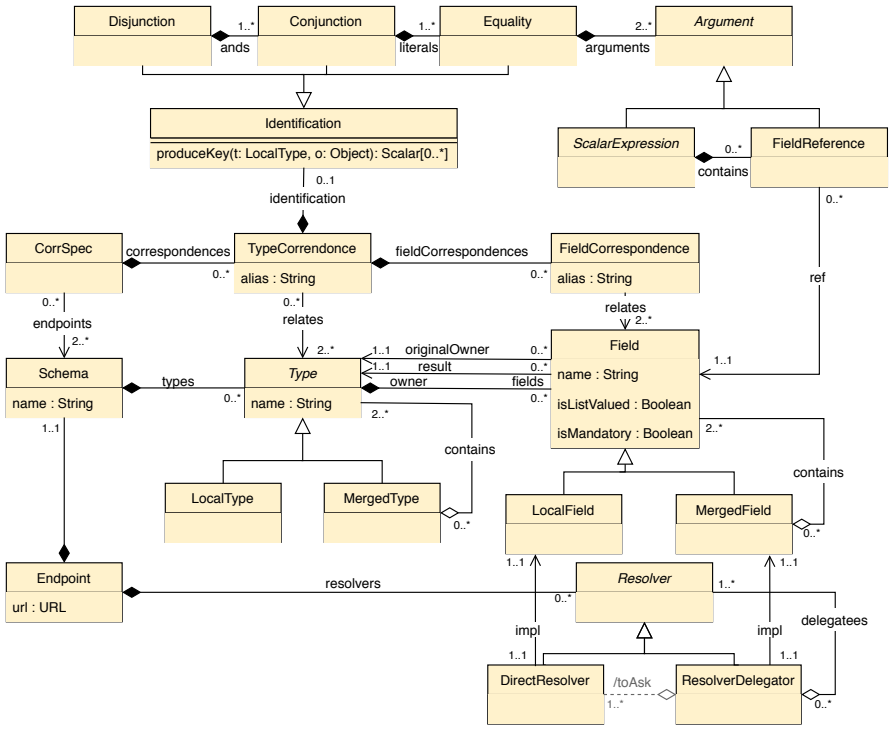

Figure 10 – Metamodel of GraphQL multimodels

containing its respective `LocalTypes`, which in turn contain `LocalFields`. A `relate`-statement over types translates to a `TypeCorrespondence`, which again contains `FieldCorrespondences` and `Identifications`. The model merging procedure can be described by an endogenous model transformation over this metamodel: For every `TypeCorrespondence` a `MergedType` is created and for every `FieldCorrespondence` a `MergedField` is created. The merge produces a new `Schema` containing all `MergedTypes` plus copies of those `LocalTypes`, which have not been merged. Finally, every `MergedField` is implemented by a `ResolverDelegator`. A `ResolverDelegator` produces its result by querying `DirectResolvers`. However, there is a technical intricacy here: due to non-intrusiveness, we cannot directly call a resolver of another endpoint, like Apollo Federation does, but instead have to contact them the regular way by sending a query. Thus, a `ResolverDelegator` has to pre-calculate the tree of `LocalFields` he has to retrieve in a query, see the derived `toAsk` association. A benefit of this approach is that it reduces to number of requests send in between the endpoints. Furthermore, a delegating resolver has to perform *consolidation*. If there exists an `Identification` on the return type, the collection of objects provided by the local resolvers has to be scanned for duplicates. This is done by creating the key for every object according to the expression defined in the `CorrSpec`, see operation `produceKey` in `Identification`. When two objects have the same key, they are merged recursively.

The solution described above has been implemented in a proof-of-concept implementation[2]. This implementation has been conducted as part of the Master thesis of the second author. All technical details can be found in his thesis [vB20]. To address

[2]The tool is available at `https://gitlab.com/olevonbargen/graphqlintegrator/-/tree/master`.

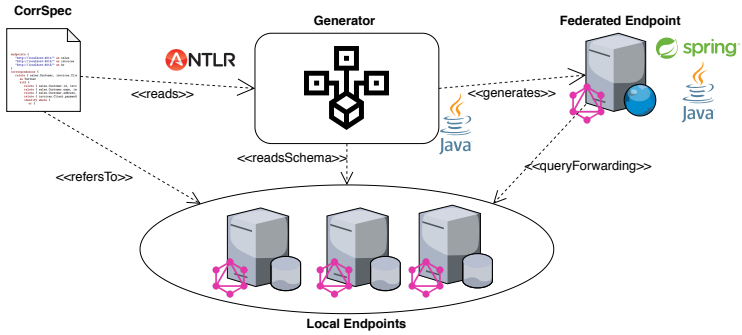

Figure 11 – Workflow of the Prototype

the technology dependency limitation (Lim#5), the tool is based on code generation. Figure 11 shows a conceptual picture of the implementation. First, we developed a Grammar for CorrSpecs (compare Figure 8) and used *ANTLR* [Par07] to generate the respective parser infrastructure. A `CorrSpec` file is fed to a Java application, which generates a Java Web application. The latter uses *Spring Boot* and *graphql-java* to leverage the technical WS-implementation details and could be easily exchanged by another technology stack. Furthermore, there are no restrictions on what GraphQL implementation has to be used in the local endpoints.

To simplify the first development iteration of the prototype, we chose to ignore correspondences on FieldArguments, which may be useful for consolidating mutations as they heavily rely on Arguments. Furthermore, GraphQL comprises a set of advanced type level features, which, for the sake of simplicity, were excluded from the discussion in section 2.1. For matters of complete presentation, we mention them briefly here: Besides `Query` and `Mutation`, there is actually a third facade type called `Subscription`. Fields of this type are intended to open a long-running channel through which the server keeps the client updated over changes. The GraphQL schema languages allows to define custom Enum Types, Union Types, or Interfaces with the typical semantics known from other programming languages. In our current prototype, `Subscription`s are explicitly not supported. The support for Enum Types, Union Types, and Interfaces is limited in a sense that it is not possible to use them in a type correspondence but one may still use them locally. Concretely, it is not possible to create correspondences for Arguments, Enum Types, Union Types, nor Interfaces.

Figure 12 shows a screenshot showing a GraphQL client querying the integrated endpoint. On the right, a small portion of the merged schema is visible, which shows the `Partner` type with all its fields. On the left, an example query involving fields from all local endpoints is formulated: It retrieves `purchases` (*Sales*), `invoices` (*Invoices*), and `worksAt` (*HR*). The response part in the middle shows two (remaining answers are collapsed) objects representing `Partners`. The first partner (`"Tabbi"`) is an employee working in the sales department, who has some purchases and invoices. The other partner (`"Beverlie"`) works in IT but has apparently never purchased anything from her employer.

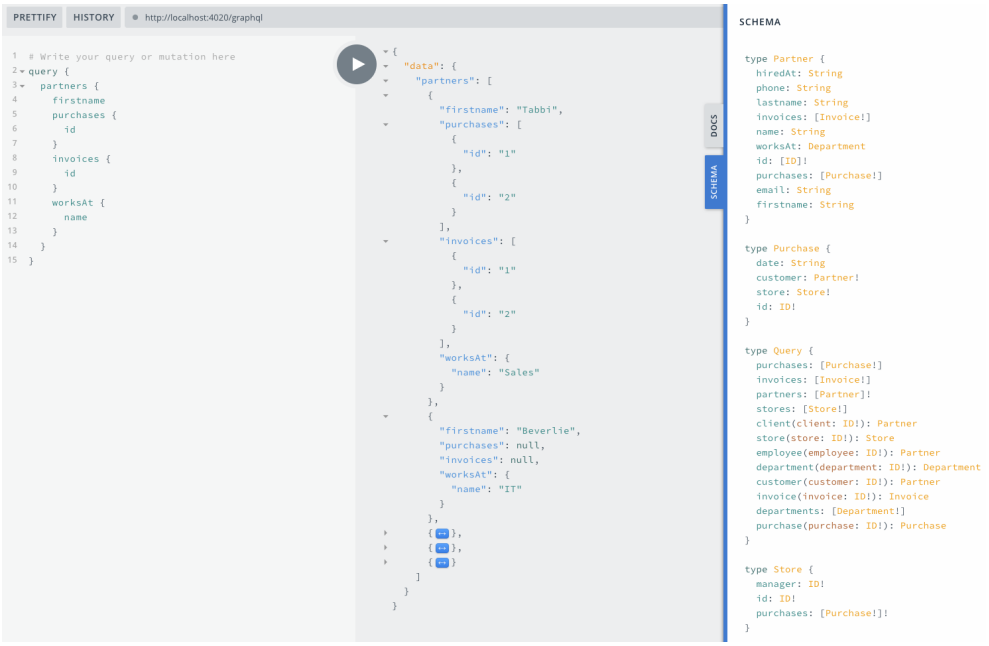

Figure 12 – Screenshot of the integrated endpoint

5 Evaluation

Table 1 provides a *qualitative* comparison between Apollo Federation and our framework: A checkmark (✓) denotes a *fully supported* feature, a checkmark in brackets indicates *limited* support, and a minus (-) indicates *missing* support. The first two rows summarize the architectural difference between Apollo Federation and our tool. Apollo Federation is based on projective multi-view modeling, assumes a unambiguous underlying system model and allows to split this definition into multiple endpoints via type extensions and keys. Our approach follows a synthetic approach, allowing to reconcile possibly conflicting views by identifying type and field definitions from existing schemas retroactively via correspondences. Furthermore, this correspondences are multi-ary compared to binary type extensions. Rows 3 and 4 show the concrete ramifications of the Apollo Federation architecture: It is intrusive and depends on a certain technology. Rows 5-7 resemble the possibilities for element identification: Both tools support alternative and composite keys but our approach additionally allows arbitrary computations on keys (e.g. concatenations). The bottommost rows (8-13) refer to technical details of the supported GraphQL functionality: Both tools support queries and mutations but neither of them supports subscriptions. Further, both tools faithfully support the multiplicities list-valued, mandatory and optional. Our solution currently lacks full support for field arguments in a sense that we cannot define correspondences upon them and we cannot define correspondences between Enums, Interfaces or Union Types yet.

Since both tools show rather big differences in terms of conception and feature support, we consider a performance comparison in the current stage less meaningful. For example, in Apollo Federation it is technically not possible to define a query methods that provides all clients, customers and employees in a single query. Thus,

Feature	Apollo Federation	Our Tool
1. Multi-View Architecture	projective	synthetic
2. Cross-Reference Arity	binary	multi-ary
3. Non-intrusive	-	✓
4. Technology agnostic	-	✓
5. Identification: Composite Keys	✓	✓
6. Identification: Alternative Keys	✓	✓
7. Identification: Key Transformations	-	✓
8. Support for Queries	✓	✓
9. Support for Mutations	✓	✓
10. Support for Subscriptions	-	-
11. Support for Multiplicities	✓	✓
12. Support for Field Arguments	✓	(✓)
13. Support for Enums, Interfaces, Unions	✓	(✓)

Table 1 – Qualitative Comparison

it is only possible to merge clients with customers or clients with employees one at a time. Still, we ran a small exploratory comparison between both tools on a 2017 MacBook Pro, running macOS High Sierra version 10.13.6, on an 2,3 GHz Intel Core i5 with 8GB of DDR3 RAM (2133 MHz)[3]. In an example dataset containing $1k$ partner records (distributed over all three endpoints) and $4k$ invoices and purchases, we ran queries asking for the invoices, purchases and the department of every partner. For $1k$ request Apollo Federation took $808.57ms$ on average to answer (min: $651ms$, max: $3473ms$, median: $775ms$) while our prototype took $962.89ms$ on average (min: $708ms$, max: $2427ms$, median: $907ms$). Given that the prototype was not specifically optimized for performance, the results show that our tool has no major disadvantage in this regard. Furthermore, it must be noted that Apollo Federation only retrieves and merge data from two endpoints for field at a time and thus returns less data. Still and in-depth performance analysis should be undertaken in the future.

6 Conclusion and Future Work

In this paper we investigated the problem of schema federation in GraphQL, a recent industrial Microservice-oriented framework, as an instance of of multi-view modeling. We proposed a solution to this problem motivated by prior theoretical work in the area [BCE+06, SKLR18] based on a DSL for correspondences definitions (cross-references), similar to the one proposed in [KG19]. Our solution is compared to an existing industrial tool called Apollo Federation, which shows that our solution is feasible, circumvents some of Apollos limitations, and appears not to be aggrieved performance-wise. Finally, our presentation of GraphQL as MBSE is something novel.

For the future, there are multiple directions: The missing features such as correspondences on field arguments, support for advanced GraphQL type system features and subscriptions have to be implemented. Furthermore, one may think of more convenience features in our DSL, e.g. for introducing new (derived) fields. When these features are implemented, we plan to perform a more meaningful quantitative comparison (than the exploratory tests we performed so far, see Sect. 5) between our tool and Apollo Federation, also comparing other aspects such as number of requests

[3]The code is available at `https://gitlab.com/olevonbargen/graphqlintegrator/-/tree/master/examples/scenario2`

and request sizes. Further, exploring how different distributed data set constellations affects the performance. This will hopefully allow another direction of our work: An outreach to the GraphQL community and Apollo developers to see how the acceptance of our work in practice will be. Concerning theoretical research there are more unresolved issues: This paper provides a conceptualization of the GraphQL framework in terms of MBSE-concepts but a full formalization is still missing. The graph-based nature promises to be well aligned with the formal framework of *attributed typed graphs (ATG)* [EEPT06], which is based on universal algebra and category theory. Thus, an alternate approach to [HP18] on formalizing the semantics of the GraphQL framework based on ATGs could be worthwhile. Another intriguing idea is to investigate how the developed framework can be exploited further in terms of *system interoperability* and verifying the *global consistency* of multiple information systems.

References

[AAE16] Nuha Alshuqayran, Nour Ali, and Roger Evans. A Systematic Mapping Study in Microservice Architecture. In *SOCA 2016*, pages 44–51, November 2016. doi:10.1109/SOCA.2016.15.

[AC08] Michał Antkiewicz and Krzysztof Czarnecki. Design Space of Heterogeneous Synchronization. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE 2007*, Lecture Notes in Computer Science, pages 3–46. Springer, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-88643-3_1.

[AK01] Colin Atkinson and Thomas Kühne. The Essence of Multilevel Metamodeling. In Martin Gogolla and Cris Kobryn, editors, *UML 2001*, LNCS, pages 19–33, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-45441-1_3.

[AL19] Preyashi Agarwal and J. Lakshmi. Cost Aware Resource Sizing and Scaling of Microservices. In *CCIOT 2019*, pages 66–74, Tokyo, Japan, September 2019. ACM. doi:10.1145/3361821.3361823.

[ARNRSG06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006. doi:10.1147/sj.453.0515.

[ATM15] Colin Atkinson, Christian Tunjic, and Torben Möller. Fundamental Realization Strategies for Multi-view Specification Environments. In *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*, pages 40–49, September 2015. ISSN: 1541-7719. doi:10.1109/EDOC.2015.17.

[BBCW19] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. A feature-based survey of model view approaches. *Software & Systems Modeling*, 18(3):1931–1952, June 2019. doi:10.1007/s10270-017-0622-9.

[BCE+06] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A Manifesto for Model Merging. In *GaMMa '06*, pages 5–12, New York, NY, USA, 2006. ACM. doi:10.1145/1138304.1138307.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2nd edition, 2017.

[CCP19] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. Multi-view approaches for software and system modelling: a systematic literature review. *Software and Systems Modeling*, 18(6):3207–3233, December 2019. doi:10.1007/s10270-018-00713-w.

[CDT18] Tomas Cerny, Michael J. Donahoo, and Michal Trnka. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*, 17(4):29–45, January 2018. doi:10.1145/3183628.3183631.

[CFH+09] Krzysztof Czarnecki, Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *ICMT 2009*, pages 193–204, 2009.

[DHI12] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration*. Elsevier, June 2012.

[DK04] Arie van Deursen and Paul Klint. Domain-Specific Language Design Requires Feature Descriptions. *CIT. Journal of Computing and Information Technology*, 10(1):1–17, October 2004. doi:10.2498/cit.2002.01.01.

[dLGKH18] Juan de Lara, Esther Guerra, Jörg Kienzle, and Yanis Hattab. Facet-oriented modelling: open objects for model-driven engineering. In *SLE 2018*, pages 147–159, Boston, MA, USA, October 2018. Association for Computing Machinery. doi:10.1145/3276604.3276610.

[EEPT06] Hartmut. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer, 2006.

[Fac18] Facebook Inc. GraphQL Specification, June 2018. URL: https://graphql.github.io/graphql-spec/June2018/.

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 02(01):31–57, March 1992. Publisher: World Scientific Publishing Co. doi:10.1142/S0218194092000038.

[FKWVH19] S. Feldmann, K. Kernschmidt, M. Wimmer, and B. Vogel-Heuser. Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering. *Journal of Systems and Software*, 153:105–134, July 2019. doi:10.1016/j.jss.2019.03.060.

[FL14] Martin Fowler and James Lewis. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html, Last Accessed: 07.02.2020, March 2014.

[FML17] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In *ICSA 2017*, pages 21–30, April 2017. doi:10.1109/ICSA.2017.24.

[FT00] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. In *ICSE '00*, pages 407–416, Limerick, Ireland, June 2000. ACM. doi:10.1145/337180.337228.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[HP18] Olaf Hartig and Jorge Pérez. Semantics and Complexity of GraphQL. In *WWW '18*, pages 1155–1164, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee. doi:10.1145/3178876.3186014.

[Jos07] Nicolai M. Josuttis. *SOA in Practice*. "O'Reilly Media, Inc.", 2007.

[KBGA15] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. A combined formal model for relational context-dependent roles. In *SLE 2015*, pages 113–124, Pittsburgh, PA, USA, October 2015. Association for Computing Machinery. doi:10.1145/2814251.2814255.

[KG19] Heiko Klare and Joshua Gleitze. Commonalities for Preserving Consistency of Multiple Models. In *MODELS 2019 Companion*, pages 371–378, September 2019. doi:10.1109/MODELS-C.2019.00058.

[KM18] Alexander Knapp and Till Mossakowski. Multi-view Consistency in UML: A Survey. In *Graph Transformation, Specifications, and Nets*, LNCS 10800, pages 37–60. Springer, Cham, 2018.

[Kü06] Thomas Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4):369–385, December 2006. doi:10.1007/s10270-006-0017-9.

[Met] Meteor Development Group Inc. Apollo federation overview. https://www.apollographql.com/docs/apollo-server/federation/introduction/, Last Accessed: 07.02.2020.

[MGdL17] Fernando Macías, Esther Guerra, and Juan de Lara. Towards Rearchitecting Meta-Models into Multi-level Models. In Heinrich C. Mayr, Giancarlo Guizzardi, Hui Ma, and Oscar Pastor, editors, *Conceptual Modeling*, LNCS, pages 59–68, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-69904-2_5.

[MRKIP+19] Suresh Kumar Mukhiya, Fazle Rabbi, Violet Ka I Pun, Adrian Rutle, and Yngve Lamo. A GraphQL approach to Healthcare Information Exchange with HL7 FHIR. *Procedia Computer Science*, 160:338–345, January 2019. doi:10.1016/j.procs.2019.11.082.

[MWR+19] Fernando Macías, Uwe Wolter, Adrian Rutle, Francisco Durán, and Roberto Rodriguez-Echeverria. Multilevel coupled model transformations for precise and reusable definition of model behaviour. *Journal of Logical and Algebraic Methods in Programming*, 106:167–195, August 2019. doi:10.1016/j.jlamp.2018.12.005.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-specific Languages*. Pragmatic Bookshelf, 2007. Google-Books-ID: 3vE5ngEACAAJ.

[RC13] Julia Rubin and Marsha Chechik. N-way Model Merging. In *ESEC/FSE 2013*, pages 301–311, New York, NY, USA, 2013. ACM. doi:10.1145/2491411.2491446.

[Ric15] Mark Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Inc, 1005 Gravenstein Highway North, Sebastopol, CA 95472., first edition edition, November 2015.

[SE05] M Sabetzadeh and Steve Easterbrook. An Algebraic Framework for Merging Incomplete and Inconsistent Views. In *RE 2005*, pages 306–315, 2005.

[SKLR18] Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. Multimodel correspondence through inter-model constraints. In *BX@<Programming>2018*. ACM, 2 2018.

[Ste20] Perdita Stevens. Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels. *Software and Systems Modeling*, March 2020. doi:10.1007/s10270-020-00788-4.

[TS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007.

[UKT+19] H. Ulrich, J. Kern, D. Tas, A. K. Kock-Schoppenhauer, F. Ückert, J. Ingenerf, and M. Lablans. QL4mdr: a GraphQL query language for ISO 11179-based metadata repositories. *BMC Medical Informatics and Decision Making*, 19(1):45, March 2019. doi:10.1186/s12911-019-0794-z.

[vB20] Ole Christoph von Bargen. Integration of Web Services and their data models with special regard to GraphQL. Master's thesis, Høgskulen på Vestlandet, Bergen, Norway, 2020. URL: https://gitlab.com/olevonbargen/graphqlintegrator/-/blob/master/thesis/Integration_of_Web_Services_and_their_data_models_with_special_regard_to_GraphQL.pdf.

[Wor07] World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) 1.2, 2007. URL: https://www.w3.org/TR/soap12/.

About the authors

Patrick Stünkel is a Ph.D. research fellow at Høgskulen på Vestlandet in Bergen, Norway. His research interests are Software (Multi) Modeling, Interoperability, System Integration and Applications of Category Theory in Software Engineering. In his thesis he investigates means for expression and restoration of consistency in multi models. Contact him at past@hvl.no

Ole von Bargen is a master student at Fachhochschule für die Wirtschaft Hannover, Germany. In parallel, he is employed at Vereinigte Hannoversche Versicherung (VHV) insurance group as an IT project manager. For his master thesis he worked at Høgskulen på Vestlandet in Bergen, Norway, where he investigated the integration of web services with special regard to GraphQL. Contact him at `ole.vonbargen@googlemail.com`

Adrian Rutle holds a PhD in Computer Science from the University of Bergen, Norway. Rutle is a professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences, Bergen. Rutle's main interest is applying theoretical results from the field of model-driven software engineering to practical domains and has expertise in the development of modelling frameworks and domain-specific modelling languages. He also conducts research in the fields of modelling and simulation for robotics, eHealth, digital fabrication, smart systems and machine learning. Contact him at `aru@hvl.no`

Yngve Lamo holds a PhD in Computer Science and is a professor at the Department of Computer science, Electrical engineering and Mathematical sciences at the Western Norway University of Applied Sciences, Bergen. His research interests span from formal foundations of Model Driven Software Engineering over applications to Health Informatics. He is leading the work package concerning the core clinical process in the Norwegian National cross-disciplinary research project INTROMAT (INtroducing personalized TReatment Of Mental health problems using Adaptive Technology) Contact him at `yla@hvl.no`

Acknowledgments We would like to thank the anonymous reviewers for their valuable feedback and input on the preliminary version, which helped to improve the quality of this paper significantly. Further, we would like to thank Harald König (`Harald.Koenig@fhdw.de`) for his input as a supervisor to the Master Thesis of the second author, which lead to this paper.