A Transformational Approach to Managing Data Model Evolution of Web Services

Luca Beurer-Kellner, Jens von Pilgrim, Christos Tsigkanos, and Timo Kehrer

Abstract—The communication of web services is typically organized through public APIs, which rely on a common data model shared among all system components. Over time, this data model must be changed in order to accommodate new or changing requirements, and the system components including the data they are operating on must be migrated. In practice, however, not all the affected components can be migrated instantly and at the same time. A common approach is to plan data model changes in a backward compatible fashion, which eventually causes serious maintenance problems and is a common cause of technical debt. In this paper, we propose an alternative solution to this problem by using a translation layer serving as a round-trip migration service which is responsible for the lossless forth-and-back translation of object-oriented data model instances of different versions. We present a framework which offers a version-aware interface definition language (IDL) for APIs, a typed JavaScript-based language for defining migration functions using the IDL definition, and a run-time environment for executing migrations. This is bundled into an integrated development environment assisting developers in implementing migration functions. From a methodological point of view, the development of round-trip migrations is supported by a catalog which comprises a set of typical data model evolution scenarios along with corresponding suitable round-trip migration strategies. We validate our framework by carrying out an extensive evaluation including a systematic assessment of expressiveness using our catalog, micro-benchmarking the performance of round-trip migrations, as well as a practical application in a case study of a real-world e-commerce web application obtained from an industrial partner.

Index Terms—Web Services, Service Data Models, Web API Evolution, Model Transformations

1 Introduction

Contemporary distributed systems are characterised by an omnipresence of web services, a paradigm that has fueled programmatic interactions between different web-facing systems. A service interface defines functionality visible to the external world and provides the means to access this functionality – such as available operations, parameters and access protocols – in a way that other software modules can determine what it does, how to invoke its functionality, and what result to expect in return. Communication within web-based services is typically organized through APIs. Those, often implemented using technologies like Apache Thrift, gRPC Remote Procedure Calls or plain REST endpoints, typically rely on a common data model shared among all system components (i.e., services).

Over time, the shared data model must be changed to accommodate new or changing requirements, and the system components including the data they are operating on must be migrated. This problem, commonly known as API evolution, is a well-known challenge, in particular for web APIs [1], [2], [3]. In practice, however, not all affected components can be migrated instantly and at the same time, particularly when web-based services and clients are developed and maintained by different teams or even different parties. Since the API's functionality is provided by a service, clients cannot just stick to an older version, which is possible in case of conventional software libraries or frameworks. This may lead to severe problems for client programmers [4]. A common workaround for this legacy problem is that API providers plan data model changes in a backward compatible fashion. However, this severely hampers flexibility when evolving the data model. Essentially, either many versions of the same data model must be maintained in parallel, or a superimposed data model unifying all versions may be used, which is hard to maintain in the long run. Both workarounds are common causes of technical debt [5]. This even poses a problem for modern API technologies like GraphQL[1], where such a data model can still be identified, although the coupling between two services is dramatically reduced.

To address this issue, we consider an alternative solution that relies on translation layers between different services and components of a distributed system to enable backwards-compatible API use. To illustrate, Fig. 1 shows typical examples of distributed systems exposing a three-tier architecture with a client, a service and a database layer. The API and its underlying data model are evolved from version 1 (red) to version 2 (purple), which may lead to several evolution scenarios. In all these scenarios, we need a translation layer (TL) which is responsible for the lossless translation of data model instances. Ideally, all components are updated simultaneously (❶); in this case we need a translation layer to migrate the existing data using tools such as Liquibase[2]. If not all components are updated, we can identify different cases, each of which requires a translation layer to migrate the data on demand (❷, ❸, ❹).

Scenario ❹ is particularly interesting: The database and service are updated, as they are maintained by the API provider. The client is not updated but it uses a library

• *L. Beurer-Kellner is with ETH Zürich, Switzerland, J. von Pilgrim is with the Hamburg University of Applied Sciences, Germany, and C. Tsigkanos and T. Kehrer are with the University of Bern, Switzerland.*

Manuscript received XXX 2021; revised XXX, 2022.

1. graphql.org 2. liquibase.com

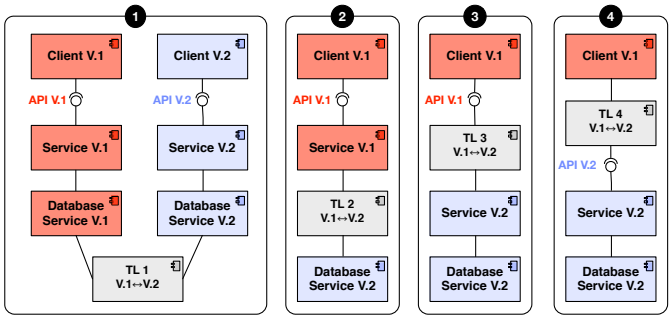

Fig. 1. An example of a distributed system in which data model evolution is supported through (round-trip) migrating data translation layers.

including the translation layer. In this case, the migration is performed by the client, which avoids any migration workload on the server side. Even though the migration is done on client side, the translation layer itself is still provided by the API provider by means of a library. Eventually, it is the API provider who knows what needs to be migrated.

Translation layers are a practical solution that can be found in real-world systems, supported by tools such as GraphQL[1] and Liquibase[2]. However, only few practical efforts have been directed at supporting the underlying problem of migrating data model instances forth and back between different model versions. To address this, we present a novel framework for round-trip migration development. Our system relies on a versioned interface definition language (IDL) and an imperative migration language, based on model transformation techniques. We implement our framework as an internal domain-specific language on top of JavaScript, well-known to many web developers.

In our framework, we focus on objects sent forth and back between components which are to be migrated. Changes of functions (i.e., methods in object-oriented or endpoints in HTTP terminology) may be handled by a translation layer as well, but are out of the scope of this paper. Similarly, protocol changes [1] (e.g., change of message format, authentication, rate limit) are also not considered, as we solely focus on the data powering the web services. Finally, we focus on a single round-trip migration at a time and do not consider concurrent operations.

Our contributions are in the form of a comprehensive framework facilitating the implementation of translation layers for the data model evolution of Service APIs. Specifically,

1) we provide a version-aware interface definition language (IDL) for object-oriented service APIs,
2) a typed JavaScript-based language for defining migration functions using the IDL definition, and
3) a run-time environment for executing migrations which can be used by a translation layer in order to realize round-trip migrations.
4) Methodologically, development of round-trip migrations is supported by a catalogue which comprises a set of typical data model evolution scenarios along with a suitable round-trip migration strategy.
5) We validate our framework by carrying out an extensive evaluation including a systematic assessment of expressiveness, micro-benchmarking as well as a practical application to the case of a data-intensive real-world e-commerce web application.

Our framework allows for flexibility when changing data models, since it does not require backward compatibility. In contrast to other approaches proposed in the literature, our framework is explicitly tailored to the practical needs of round-trip migration development in the context of web services. Furthermore, we provide methodological support for migration scenarios, in turn elicited from previous research on edit operations by Hermannsdörfer et al. [6] within metamodel evolution, in the form of a patterns catalog. Each consists of a basic edit operation and a suitable migration in a translation layer using our framework.

The remainder of this paper is organized as follows. In Sec. 2, we motivate our work and research goals by introducing a running example used throughout the paper. In particular, we derive design requirements for our framework that aims at supporting the effective and flexible development of round-trip migrations. To fulfill these requirements, Sec. 3 introduces a version-aware interface definition language (IDL) which lays the foundations for the migration framework described in Sec. 4. Last, Sec. 5 presents our validation, including a catalog of round-trip migration scenarios to assess expressiveness, the results of our micro-benchmarking experiments as well as the implementation of a translation layer using an industrial case study. We discuss related work in Sec. 6, and conclude along with an outlook on future work in Sec. 7.

2 Background and Motivation

In this section, we first introduce a simple example of object-oriented data model evolution, along with a discussion of according round-trip migration scenarios of data model instances. Thereby, we introduce basic terms and motivate our work, before we distill specific design requirements for our round-trip migration framework that aims at supporting the effective and flexible development of round-trip migrations.

2.1 Motivating Example

Fig. 2 shows a typical evolution step of an object-oriented data model (using UML class diagram notation). The example is based on the so-called *Families to Persons Case* [7], a well-known benchmark in the related field of bidirectional model transformation (see discussion of related work in Sec. 6). We slightly adapted and reduced the original benchmark here to keep the example as small as possible while still serving as a motivation for our work. In essence, a `Family` comprises an arbitrary number of family `members`, and members are represented as objects of class `Person`, which in turn defines two fields of type String. In the original version of the data model on the left, called M_1, both fields are mandatory. The value of the field `name` shall represent the given name (i.e., first name) of a person. The gender of a person may be unspecified, which shall be represented by an empty string value (written `" "` in the sequel) of the field gender. Two changes have been applied to the data model to become the revised version on the right, called M_2. First, the field `name` has been renamed to `firstname` in order to account for the fact that it intends to represent the given name of a person. Second, the field gender has been declared to be optional (indicated by the notation `[?]`); the rationale behind this change is that the value of this field may be omitted for privacy reasons.

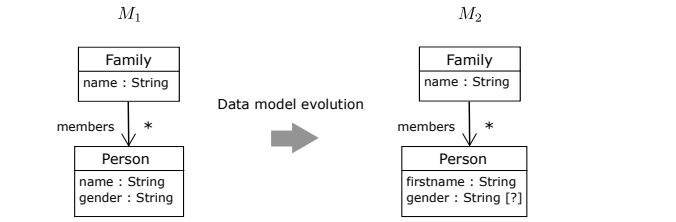

Fig. 2. Motivating example: Evolution step of an object-oriented data model which is part of the API of services in a distributed system.

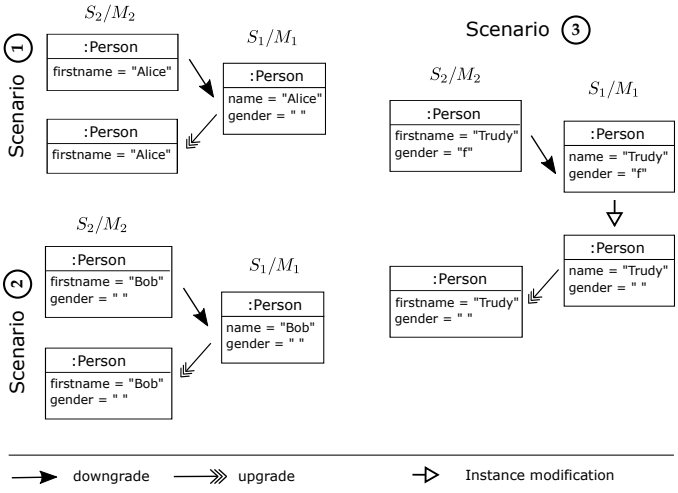

Fig. 3. Motivating example: Three different round-trip migration scenarios of object-oriented data model instances.

Now, we assume two communicating services, called S_1 and S_2, where S_2 has been already migrated to data model version M_2 while S_1 is still based on M_1. This leads to a couple of interesting round-trip migration scenarios, three of which are illustrated in Fig. 3. In all three scenarios, service S_2 sends an object of type `Person` to service S_1, thus the `Person` object needs to be downgraded to be a valid instance of M_1. Later on, service S_1 sends back the object, which needs to be upgraded from M_1 to M_2 again. Given the information that the field `name` has been renamed to `firstname` (i.e., the fields `name` and `firstname` are corresponding fields in data model versions M_1 and M_2, respectively), the value of this field can be simply copied forth and back in all the migrations (both upgrades and downgrades) shown in Fig. 3. The more interesting aspect in our example is how deal with the field `gender`, which we will consider in more detail for each of the three scenarios in the remainder of this section.

In the first round-trip migration scenario ① (on the upper left), the `Person` object sent by service S_2 does not expose any gender information. This is possible since the field `gender` is declared to be optional in data model version M_2. In M_1 on the contrary, the respective field is mandatory. Therefore, the downgrade migration must somehow account for a `null`-value, e.g. set the value of the mandatory field to the default value of its respective data type (i.e., the empty String in our example). Later on, when the `Person` object is sent back to S_2, it is to be upgraded by restoring the respective `null`-value.

In the second round-trip migration scenario ② (on the lower left), the `Person` object sent by S_2 *does* expose gender information, which is deliberately unspecified (as opposed to the `null`-value in the first scenario which hides gender information for privacy reasons). In this case, the empty String value of the field `gender` may be simply copied in both the downgrade and the upgrade migration.

A challenge arises when comparing the first and the second scenario. When upgrading a `Person` object from M_1 to M_2, the empty String value of field `gender` is simply copied in the second scenario, while the field is re-initialized with a `null`-value in the first scenario. This means that, in terms of the respective upgrade migrations, we cannot *blindly* copy the value of the field `gender`, but the suitable value can be only inferred by inspecting the previous state of `Person` before it has been sent (and thus downgraded) to S_1. Notably, in both scenarios, the initial `Person` sent by S_2 is equal to the `Person` obtained after round-trip migration. We call such a migration a *successful round-trip migration*.

In practice, however, round-trip migrations as introduced above seldomly occur. Typically, a service will not directly return an instance it just received but rather apply some modification before returning it. This is illustrated by the third round-trip migration scenario ③ shown in Fig. 3 (on the right). Here, the gender of the `Person` object sent by S_2 is changed to unspecified (represented by the empty String value) by service S_1. When sending the object back to S_2, we keep this information in terms of the upgrade migration by copying the empty String value. As opposed to the first and second scenario, we do not restore the proper value of the field `gender` from the previous M_2 state of the `Person` object, but account for the modification that has been performed by S_1. In other words, the result of upgrading the modified object is *not* expected to be equal to the original one, but the upgraded object rather reflects the performed modification. In general, when an instance is modified by a service that works on a particular version of the data model, we need to account for this modification by the corresponding co-modification in the other version of the data model (in our example, modification and co-modification are identical). We call migrations that account for such modifications *successful round-trip migrations with modification*.

2.2 Research Goals

Our overall goal is to support data model evolution by using a translation layer serving as a round-trip migration service which is responsible for the lossless forth-and-back translation of object-oriented data model instances of different versions. A translation layer is called successfully round-trip migrating (with modification) if, for all data model instances being sent forth and back between services based on different data model versions, the respective round-trip migrations are successful (with modification).

As already indicated by our simple example, specifications of the required upgrade and downgrade migrations cannot be inferred automatically, but they need to be provided by developers who are aware of the semantics of the underlying data model (e.g., the different meanings of the empty String value of the field `gender`) and its changes during an evolution step (e.g., the fact that the differently

named fields `name` and `firstname` are corresponding fields that convey the same information). Based on this perspective, we aim at providing a framework that supports the development of translation layers, and its design shall meet the following design requirements.

First of all, within migration specifications, a mechanism is needed to track and provide control over different versions of a data model at the API level (i.e., a version-aware IDL). Since our focus is on the evolution of object-oriented data models and the migration of its instances, we derive the following design requirement that a migration framework should fulfill:

(DR1) Interface definition should support versioning at the model level.

Second, translation layers developed using our framework shall be successfully round-trip migrating (with modification). Ultimately, for the same reasons for which an automated inference of translation layers is impossible, this correctness property of translation layers is left to the discretion of developers using our framework. It must be assured using classical quality assurance techniques known from software engineering, notably software testing, which are out of the scope of our work. However, our framework must enable assessing all the relevant information which is necessary to implement strategies which account for the particularities of round-trip migrations (as discussed for our motivating example), which we compile into the following functional design requirement:

(DR2) The framework should enable the development of translation layers which are successfully round-trip migrating.

On top of a version-aware IDL, this includes a migration specification language which is easy to understand for a wide range of practitioners and which makes all the relevant information (such as version and traceability constructs) easily accessible.

Lastly, although there is no fully automated way of inferring migrations, it is evident from our motivating example that, in practice, data model evolution and according round-trip migration scenarios may follow certain recurring *patterns* (e.g., creating or deleting fields). Based on this observation, we derive the following requirement:

(DR3) Developers shall be guided in the development of migration functions by a catalog of frequently occurring scenarios.

In essence, the scenarios in such a catalog serve as reusable design patterns for developing round-trip migrations. The patterns do not limit development of migration functions, but are intended to provide exemplary or template solutions to typical migration problems encountered in practice that developers may freely utilize.

3 Version-Aware IDL

We first tackle requirement (DR1), as it is fundamental to defining the base language for migrations development – the DSL advocated for this purpose is an internal DSL. It is implemented on top of a general purpose language (the host language), and reuses its infrastructure (e.g., concrete syntax, type system and run-time system), which is extended with specific constructs.

```
ClassDeclaration:
    'class' Identifier Version?
    TypeVariables? ClassExtendsClause?
    Members

VarStmt: 'var' VarDecl (',' VarDecl)* ';'?
VarDecl: Identifier ColSepTypeRef?

ColSepTypeRef: ':' TypeRef

TypeRef: Identifier Version?

NewExpression = 'new' MemberExpression
    TypeArguments? ( '(' Arguments? ')' )?
MemberExpression = ... | TypeRef | ...

Version: '#' INTEGER
```

Listing 1. Simplified grammar snippet with versioning.

```
class Family#1 { members: Array<Person> ❶ }

class Person#1 {
  name: string
  gender: string
}
class Person#2 {
  firstname: string
  gender?: string
}
```

Listing 2. Example of a simple N4IDL definition with two classes in two versions.

Specifically, we introduce a version-aware IDL called N4IDL. N4IDL is built on the foundations of the typed ECMAScript variant Eclipse N4JS[3], providing a static type system similar to Java. We motivate the design decision for this with the wide-spread usage of JavaScript in both frontend and backend of many web applications. The syntax of N4IDL is based on JavaScript with type annotations, using the notation `<variable>:<type>` as proposed in ECMAScript 4 [8] and used by, e.g., the typed JavaScript implementation TypeScript[4]. Since we only define the API, the language omits function or method bodies. This is similar to type definition files as known from TypeScript for typing plain JavaScript files. Besides classes (introduced in ECMAScript 2016 [9]) we add the notion of fields, enumerations, interfaces and annotations similar to Java.

Syntactically, we introduce versions as type name suffixes, separated by a hash character (`'#'`). Listing 1 is a simplified snippet taken from the EBNF grammar: The rules coloured in magenta stem from N4JS, extending the original ECMAScript language with static types. The N4IDL versioning support is added by the red coloured rules. As we can see from the grammar the extensions used for versioning are minimal from a syntax point of view.

For versioning, we rely on a mechanism similar to the Git version control system[5], except that we adopt types instead of files as the unit of versioning and consecutive integer numbers instead of commit IDs for identifying versions, respectively. That is, there is a full API available for each version, even if not explicitly defined for all types.

As an example of this versioning scheme, consider Listing 2. Here, we use three N4IDL type definitions to model the Families and Person scenario as discussed in Sec. 2. We explicitly define types `Family` and `Person` in version 1, and only `Person` in version 2. However, `Family` is implicitly defined in version 2 as well, it is just assumed to be the same as version 1 and does not need to be defined

3. eclipse.org/n4js/ 4. typescriptlang.org 5. git-scm.com

```
@Migration
function upgradeFam(f1: Family#1): Family#2 {
  let f2 = new Family#2();
  f2.members = migrate(f1.members); ❶
  return f2;
}
```

Listing 3. Example of a simple migration comprising a migration call. The migration of an array automatically invokes the migration for each element of the array.

explicitly. To achieve this, the type `Person` referenced by the field `Family.members` has no explicitly defined version ❶. Instead, in the context of `Family#1`, the referenced type automatically binds to `Person#1`, and in the implicitly defined `Family#2`, the referenced type binds to version 2.

4 Round-Trip Migration Framework

The IDL introduced in the previous section supports the definition of versioned types. This already provides a more fine-grained versioning model as compared to mainstream version control systems. It allows versioning on the level of types rather than on a per-file basis. To come up with a fully equipped round-trip migration framework, however, we also need to operationalise the concept of migration functions as introduced in Sec. 2 and distilled in (DR2) .

Formally, a migration function can be considered to map a complete data model instance from one version to another. However, in order to decrease the complexity of migration code and to support reuse and modularity, we chose a different level of granularity for the implementation of our framework. Instead of leaving the migration developers with the task of migrating a complete data model instance, they are supposed to provide migration functions on a type level. These so-called *N4IDL migration functions* are a specific kind of annotated JavaScript functions. In contrast to their conventional counterparts, they allow for accessing the migration context at run-time. Furthermore, a dedicated migration engine controls their execution, which allows to automatically resolve circular dependencies among migrations. For brevity, we will refer to N4IDL migration functions as *migrations*.

4.1 Migrations and Migration Calls

An (N4IDL) *migration* is a global function which migrates between two versions of a type. Migrations are unidirectional and implemented imperatively. A migration from one version of a type to another is declared as an annotated function (keyword `@Migration`) on the top-level of an N4IDL file, as shown in Listing 3. In this example, the migration `upgradeFam` migrates instances of type `Family` in version 1 to type `Family` in version 2. Migrations are *version-aware*, i.e., they can refer to specific versions of a type. To do so, the hash character '#' may be used as part of type and constructor references. In general, migrations are responsible for instantiating the returned instances. This way, migrations maintain full control over the instantiation process but can also call additional library functions that perform preliminary automatic migration work (e.g. automatically copy values of fields with the same name).

An important design goal of our framework is modularity of migrations on a type level. More specifically, we want to avoid redundancy in migration code and allow developers to reason about the migration of a specific type in isolation. Therefore, we allow to delegate the migration of referenced objects from within a migration, while the migration itself only has to be concerned with the details of migrating the passed object(s). Using this mechanism, modularity is established by reducing the responsibility of a migration function to local fields, while the migration of referenced objects may be delegated. In Listing 3, for instance, the migration `upgradeFam` delegates the migration of the `Person` objects representing family members at ❶.

The use of the `migrate` keyword is referred to as a *migration call*. It differs from a conventional function call in three important aspects:

1. Binding: A migration call is always bound to the most specific migration among the set of declared migrations with subtype-compatible parameter types. This is implemented using dynamic dispatch [10], a well-known method in object-oriented programming systems, where the binding is based on the run-time type of the migration arguments.

2. Memoization: Multiple migration calls issued for the same subgraph of the migration input will only be executed once. All subsequent calls within the same migration task will return the cached result of the first invocation (see Sec. 4.3).

3. Fulfillment: Unlike regular function calls, a migration call may not always directly return the corresponding migrated object but a proxy instead. Proxies are guaranteed to be replaced by the actual migration results once all migrations triggered in the current migration task have completed. This allows the migration engine to automatically resolve circular dependencies among migrations (see Sec. 4.3).

4.2 Context Information in Migrations

A critical feature to support successful round-trip migrations is the ability to use what we refer to as *context information*. As we have seen in the Family/Person example introduced in Sec. 2, two kinds of context information are key, namely to (i) access the previous state of an object in the other revision, and (ii) to check whether a modification has been performed in the current revision. To accommodate this, N4IDL migrations may access additional context information via the `context` identifier; the former information may be accessed through transparently-managed *trace links*, and the latter information is provided through what we call *modification detection*.

Trace Links. To allow for accessing a previous revision of an instance during a later stage in the round-trip migration, the run-time implicitly establishes *trace links* between the input and return arguments of a migration call. These trace links allow to recover potentially lost information from a previous revision of an object, e.g., when upgrading an object that has been downgraded in an earlier migration. To illustrate the usage of this tracing mechanism, Listing 4 shows a possible upgrade migration for the Family/Person scenario. Here, by using trace links, the migration recovers an earlier value of the field `gender`. In a round-trip, an instance of `Person#2` is first migrated to version 1. This downgrade migration implicitly creates a trace link between the original and the migrated instance. Later on, the instance is migrated back to version 2 via the shown `upgrade` migration. During the upgrade, it is now possible to access

```
@Migration
function upgrade(p1: Person#1) : Person#2 {
  let p2 = new Person#2();
  p2.firstname = p1.name;
  // use traces to restore p2.gender
  let t = context.getTrace(p1)[0] as Person#2 ❶
  p2.gender = t ? t.gender : p1.gender;

  return p2;
}
```

Listing 4. Accessing a trace link via the migration context.

```
@Migration
function upgrade(p1: Person#1) : Person#2 {
  let p2 = new Person#2();
  // use traces to restore p2.gender
  (...)
  // check for modification
  if (context.isModified(p1, "gender")) {
    p2.gender = p1.gender;
  }
  return p2;
}
```

Listing 5. Checking for the modification of a field value via the migration context.

the previous revision of the object through the trace link that was created when upgrading ❶. This allows the migration to recover the earlier value of `gender`, if required. Also note the notation `"[0]"`, which specifies which of the inbound trace links to follow. In cases of migrations with multiple parameters, the number of incoming trace links may be greater than one.

Modification Detection. Although we can retrieve a previous revision via a trace link, a modification of an instance may have corrupted this relation on a semantic level, i.e., the previous revision of an object may no longer provide an up-to-date representation of the instance. An example of this is scenario ③ of our running example (cf. Fig. 3), where `gender` is modified in version 1. As a consequence, it no longer suffices to restore an older value from a previous revision. Instead, our framework provides a modification detection feature as illustrated in Listing 5. In addition to using trace links, the migration can check for a modification of `gender` and adapts its migration strategy accordingly. The `isModified` method indicates whether a field has been modified in the current version of a migrated instance. For a more detailed discussion of a migration that relies on both trace links and modification detection, we refer to Sec. 5.1.2.

These two types of context information are important features for implementing round-trip migrations. Trace links allow us to inspect previous revisions of the migrated instances, and modification detection gives us further insight on the explicit modification intent. The actual complexity however, lies in using this information to correctly handle instance modifications that are performed in the current model version. As pointed out, accessing the previous revision via a trace link potentially means accessing an outdated instance, since modifications only apply to the current model version. A successful round-trip migrating strategy (cf. Sec. 2) must map the modification intent back to the original model version. This is discussed in greater detail in the scenario catalog presented in Sec. 5.1.

Finally, note that a migration context as such can be made fully portable and passed along network requests as an additional payload. This is especially relevant for scenarios where a translation layer may be a service on its own. Bundling the context with requests increases the size of the overall payload, but it also avoids the need for elaborate garbage collection schemes and session management as part of the translation layer.

4.3 Execution of Round-Trip Migrations

Most often, we not only want to migrate a particular object, but a larger network of objects connected through references, i.e., the transitive closure of the object which is to be migrated. Migrating such an object graph includes determining the order in which migrations are executed. While a basic order of execution is given by the nesting of migration calls, we need to ensure during execution that multiple references to the same subgraph (cf. *Memoization*) and cyclic dependencies (cf. *Fulfillment*) are resolved correctly. For this, we differentiate three different types of object graph expected as migration input:

1. Object Trees. The input object graph has the properties of a directed tree. Since there is exactly one path from some root node to any other node of the graph, all objects can be migrated by directly fulfilling all migration calls.

2. Directed Acyclic Object Graphs. The input object graph is a directed acyclic graph. That is, there may be nodes which can be reached from the root node by more than one path and whose duplicate migration must be prevented. Multiple migration calls with the same input must therefore return the same instance (cf. *Memoization*). This is achieved by caching intermediate migration results by their arguments.

3. Cyclic Object Graphs. The input object graph is a directed graph with cycles. The cycles in the graph translate to cyclic dependencies among migrations. In particular, a migration may depend on its own output and we therefore cannot guarantee the immediate fulfillment of all migration calls. Instead, our framework detects such cases and returns a proxy result. These proxy results allow us to break cyclic dependencies among migrations, and the case of migrating cyclic object graphs can be reduced to case 2 of acyclic graphs, as described above. To restore the cyclic structure once all migrations have returned, the migration engine replaces the proxies by a reference to the corresponding part of the migration output. This guarantees the eventual restoration of the original cyclic structures in the migration output. However, this approach imposes a restriction on migrations operating on a cyclic object graph. In the case of cyclic dependencies among migrations, operations on the result of migration calls will throw a runtime exception if the corresponding migration result is a proxy. To account for this, developers should anticipate whether they are dealing with cyclic structures and adapt their code accordingly.

To summarise, our migration engine employs the following strategy. Given a migration call for a specific subgraph of the migration input, the following cases are considered based on the current state of the overall migration task:

- *Return proxy*, if the specified subgraph is currently being migrated.
- *Return cached result*, if the specified subgraph has already been fully migrated.
- *Trigger the migration*, otherwise.

The initial migration request to the translation can be considered as a root migration call. Developers using a translation

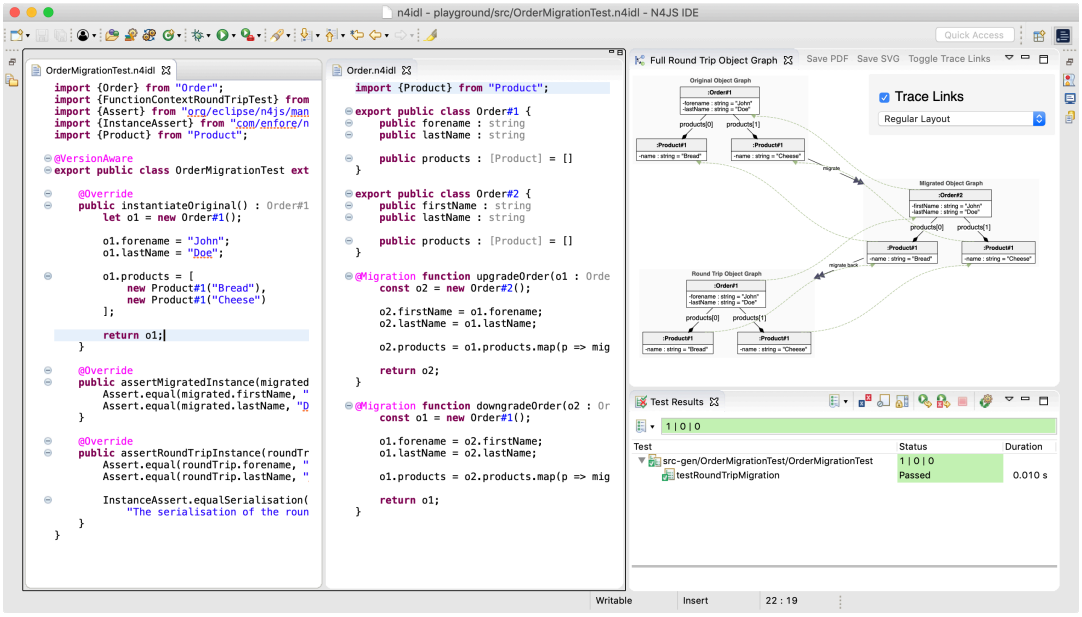

Fig. 4. The N4IDL development environment demonstrating the use of unit testing as well as the visualisation of round-trip migrations.

layer specify an entry point to the object graph to migrate. Starting with such an entry point, migration calls are processed as described until all subsequently issued migration calls are fulfilled. Last, a post-processing step replaces proxy results by the corresponding parts of the migration result, re-establishing potential reference cycles.

4.4 Semantics and Implementation

Fig. 5 illustrates key components as well as the corresponding information flow. Semantically, N4IDL is largely identical to ECMAScript. Just as in N4JS, all type expressions (used for static type information) are simply removed during compilation. Version information is compiled to special identifiers (e.g. `B#2` is transformed to `B_2`). This is also done for references omitting the version information (where the version is required and derived from the context) and for implicitly defined versions.

For execution of migration function bodies, the semantics also correspond to those of regular top-level functions in ECMAScript. However, to enable automatic selection and dispatching of the correct migration functions, additional code is generated which registers all migration functions with the framework's runtime library. This runtime library implements the execution model of migration functions as described above, supporting migration calls, trace links and modification detection via the designated `migrate` and `context` identifiers respectively. To resolve migration functions, the library implements dynamic dispatch [10],dynamically binding a migration call to the best matching migration function (see step "1. Binding" in Sec. 4).

The runtime library itself is also implemented in ECMAScript, meaning the resulting bundle of runtime library and translation layer can be executed using any JavaScript interpreter. If the translation layer is part of the client (cf. Fig. 1, ❹), it can be executed in the browser. If the translation layer is deployed on the server side (cf. Fig. 1, ❶ - ❸), it can be executed by the Node.js runtime. An architectural overview is illustrated in Fig. 5.

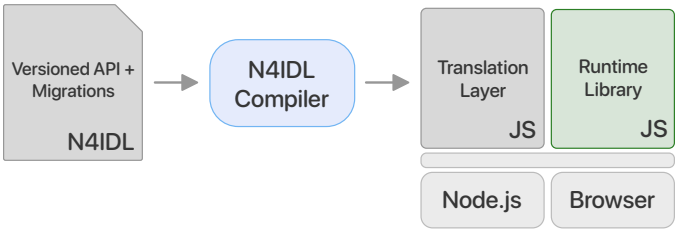

Fig. 5. The N4IDL compiler compiles a versioned API and migration code to ECMAScript. With the runtime library, the resulting translation layer can then be deployed to any JavaScript-capable environment.

4.5 Tool Support

As a derivative of the general-purpose programming language N4JS, N4IDL comes with a fully-featured IDE based on the Eclipse Platform[6]. This enables features such as syntax coloring, hyperlinks, content proposals and an incremental builder. We also extended the Eclipse N4JS transpiler by compilation support for N4IDL to ECMAScript, which allows us to execute migrations. Furthermore, it fully integrates with the unit testing capabilities of N4JS, allowing to test migration code systematically.

We further provide a designated view for the visualization of round-trip migrations to support developers in debugging their translation layer implementations. The view visualises object graphs and according trace links in different stages of a round-trip migration. Fig. 4 illustrates the N4IDL development environment, demonstrating the use of unit testing as well as the visualisation of round-trip migrations within the development process.

5 Validation

To demonstrate the effectiveness of our round-trip migration framework we perform an evaluation with respect to three core aspects: (1) We implement translation layers for an extensive catalog of evolution scenarios introduced by Hermannsdörfer et al. [6]. This part of our evaluation

6. eclipse.org/eclipseide/

TABLE 1
An overview of all round-trip migration scenarios in our catalog.

No.	Name	Description	Trace Links	Mod. Detec.	Migration Execution Time
1	Rename Field	The name of a field changes.			82.5±16.263ms
2	Create/Delete Field (independent field)	A new functionally independent field is added/removed to/from a class of the data model.	✓		81.0±8.660ms
3	Create/Delete Field (dependent field)	A field is removed from a class of the data model (or added respectively). The field is functionally dependent on other still-existing fields.	✓		66.0±5.657ms
4	Create/Delete Reference	A field of reference type is removed from a class of the data model (or added respectively).	✓		58.0±0.000ms
5	Declare Class as Abstract	A class is declared abstract/concrete.	✓		91.0±29.473ms
6	Add/Remove a Supertype	A new super type is declared for a classifier (or removed respectively).	✓		29.5±0.707ms
7	Generalize/Specialize Field Type	The type of a field is generalized to a super type or specialized to a subtype respectively.	✓		66.2±21.194ms
8	Change Field Multiplicity: 0..1 - 1	The multiplicity of a field is specialized from multiplicity exttt 0..1 to exttt 1 or generalized from exttt 1 to exttt 0..1.	✓	✓	21.0±2.828ms
9	Change Field Multiplicity: 0..n - 0..1	The multiplicity of a field is generalized from 0..1 to 0..n or specialized from 0..n to 0..1.	✓	✓	65.5±22.643ms
10	Change Field Multiplicity: 0..n - 1	The multiplicity of a field is specialized from multiplicity exttt 0..n to exttt 1 or generalized from exttt 1 to exttt 0..n respectively.	✓	✓	52.0±7.251ms
11	Pull Up / Push Down Field	A field is pulled up into a superclass (or pushed down respectively).			18.5±2.121ms
12	Split/Merge Type	Based on specified criteria, instances of a type of one model version, translate to different (unrelated) types of the other model version.	✓		46.5±6.364ms
13	Specialize/Generalize Superclass	The super type of a class is changed to one of the super type's subclasses/superclasses.	✓		20.5±4.950ms
14	Extract/Inline Superclass	A new superclass is extracted from the set of fields of an existing type.	✓	✓	35.2±8.221ms
15	Fold/Unfold Superclass	A new superclass is declared for a type. Common fields of the superclass and the type are then removed (folded into superclass).			19.5±3.536ms
16	Extract/Inline Subclass	A selection of fields is extracted into a new subclass (or inlined).	✓	✓	44.7±7.506ms
17	Extract/Inline Class	A selection of fields is extracted into a new delegate class.			18.5±2.121ms
18	Fold/Unfold Class	A selection of fields is folded into an existing delegate class.			20.5±6.364ms
19	Collect Field over Reference	A field is collected/pushed over a reference.	✓	✓	36.2±14.523ms
20	Split/Merge Fields	A type is split by moving its fields to two new types and correspondingly replacing all references to it by references to the new types.			24.0±6.245ms

demonstrates that our framework can effectively handle a wide range of realistic evolution scenarios. Given this set of implemented catalog migrations, (2) we perform micro-benchmarking of running round-trip migrations, providing an intuition of how much runtime overhead a translation layer introduces. Last, (3) we demonstrate applicability in practice over a characteristic case sourced from an industrial partner, which entails the evolution of a data-intensive real-world e-commerce web application.

5.1 Catalog of Migration Scenarios

To assess the effectiveness of our framework in a diverse range of evolution scenarios, we have successfully implemented a large catalog of what we call round-trip migration scenarios. Each scenario consists of a basic edit operation on a data model, a discussion of the change induced by the edit operation, and suitable N4IDL migrations that implement a corresponding translation layer using our framework. Where necessary, a scenario also includes a discussion of different variations and potential modifications inbetween migrations. An overview of our catalog is presented in Table 1. Each scenario has (i) a unique identifier, (ii) a name, (iii) a description of the corresponding data model edit operation(s), and an indicator whether (iv) trace links or (v) modification detection are used for employing a suitable round-trip migration strategy. Due to space limitations, we only discuss one selected scenario here. The interested reader is kindly referred to the online appendix[7].

5.1.1 Evolution Scenarios

The evolution scenarios comprised by our catalog are based on the set of edit operations collected by Herrmannsdörfer

7. The entire catalog is available as a technical report in supplementary material: github.com/AnonAuth/RoundTripMigrations

```
class A#1 { a : string }
class A#2 { a? : string }
```

Listing 6. The original and revised data model of scenario 8.

```
@Migration function upgrade(o1: A#1) : A#2 {
  let o2 = new A#2();
  let prevRev = context.getTrace(o1)[0] as A#2
  // If "A" has not been modified,
  // and a previous revision can be obtained
  if (prevRev != undefined &&
    !context.isModified(o1, "a")) {
    o2.a = prevRev.a; // reuse the previous value of
        "a"
  } else {
    o2.a = o1.a; // otherwise copy over "o1.a"
  }
  return o2;
}
@Migration function downgrade(o2: A#2) : A#1 {
  let o1 = new A#1();
  // use default value in case "o2.a" is null
  o1.a = o2.a || "default";
  return o1;
}
```

Listing 7. The migration functions of scenario 8.

et al. [6]. Although their work addresses model migration in response to metamodel evolution (see Sec. 6.3), the presented edit operations on metamodels correspond to those on object-oriented data models. As a consequence, we regard their collection of evolution scenarios as a useful external benchmark to validate our framework.

5.1.2 Detailed Description of a Selected Scenario

As an example, we discuss round-trip migration scenario 8 *Change Field Multiplicity: 0..1 - 1* of our catalog. This also demonstrates how to use our framework to implement a concrete migration strategy. In the scenario, the multiplicity of a field is changed from 1 to 0..1 or vice versa, thus allowing (multiplicity of 0..1) or disallowing (multiplicity of 1) null values. An example of this is given by the N4IDL declarations in Listing 6, where version 1 of class `A` declares its field `a` to be mandatory while it is optional in version 2. Note how this scenario directly relates to parts of our initial example in Fig. 2, where field `gender` is declared optional.

The respective migration functions are shown in Listing 7. In an $M_1 \mapsto M_2 \mapsto M_1$ round-trip migration, we must ensure that a potential `null`-value of field `a` in version 2 is translated to a default value in version 1. Similarly, we must ensure that an $M_2 \mapsto M_1 \mapsto M_2$ round-trip migration restores the `null`-value in case the instance has not been modified in version 1. To realise this, we must employ the modification detection feature of our framework to differentiate the case of the actual value "default" and the *default value* "default". The former should be migrated as is, while the latter should be translated to a `null`-value.

5.1.3 Catalog Results

Given the catalog of evolution scenarios introduced by Herrmannsdörfer et al. [6], we were successful in applying our framework to implement translation layers for a diverse set of scenarios, as listed in Table 1. All of the implemented translation layers implement the idea of a successful round-trip migrations (with modification) as introduced in Sec. 2.1. We believe this serves as strong evidence that our framework is indeed effective in addressing the problem of round-trip migration. We also consider this as confirmation that our theoretical model of round-trip migrations is feasible in the context of a wide range of different evolution scenarios. Lastly, given the overview in Table 1, we can also see that the notion of trace links as well as modification detection are essential features of our framework, required across a wide range of different scenarios.

5.1.4 Pattern-Oriented Migration Development

Next to serving as an evaluation of our framework, we consider our catalog of migration scenarios as a useful resource for migration development. Rather than conceiving migrations in an ad-hoc way, developers can reuse the provided solutions to recurrent migration problems. This idea of pattern-oriented development is well-known in the context of software design and we see great potential in applying a similar approach to migration development.

However, please note that we do not claim our catalog of evolution scenarios to be complete. This would require that (i) any difference between two versions of a data model can be decomposed into edit operations comprised by the catalog and (ii) that the migration strategies for those edit operations can be composed into a migration strategy which is successfully round-trip-migrating at the composite level. As argued by [6], such completeness is hard to achieve as higher-level changes often entail migration strategies that differ from the simple concatenation of the strategies that fit their elementary edit operations. This is particularly the case for overlapping edit operations, an example of which will be considered in the case study of Sec. 5.3.

While some scenarios may be applicable exactly as presented, others may be more suitable to demonstrate abstract aspects. As will be evident within our case study (Sec. 5), it often helps to first consider related round-trip migration scenarios before implementing a concrete migration strategy. Generally, the catalog remains an open collection of elementary round-trip migration scenarios. In the future, it may be extended by additional scenarios based on practical experience or new features in the underlying IDL.

5.2 Micro-Benchmarking of Catalog Scenarios

Our work mostly relates to the use of translation layers as a middleware between services, and we seek to gain insights on the overhead of using such a translation layer implemented using our framework. In practice, we can expect situations in which we migrate a large number of small instances. This is typical in the web service setting, due to bandwidth limitations. Micro-benchmarking – performing tests designed to measure a very small performance unit – fits our context particularly well, since scenarios represent well-defined units, while measurement entails evaluating execution time. To obtain a representative set of scenarios, we rely on our catalog discussed in the previous section.

We measure the average migration execution time of all the translation layers implemented in our catalog. All experiments were run on a 2020 Apple MacBook Pro with a 2.3 GHz Quad-Core Intel Core i7 processor and 32GB of main memory, using Node.js 16.4.2. The benchmarking script relies on Node.js's `process.hrtime` to measure execution time and first runs 50 warm up iterations per RTM, to prevent just-in-time compilation overhead during the actual benchmarking. Further, we only measure the

TABLE 2
Occurrences of observable model changes and related scenarios of our round-trip migration catalog.

Σ	Change	Related Scenarios
4	Rename Field	ID 1: Rename Field
2	Change Field Multiplicity from 0..1 to 1	ID 8: Generalize Field Multiplicity from 0..1 to 1
1	Change Field Multiplicity 0..n to 1	ID 10: Generalize Multiplicity 0..n to 1
3	Add field (functionally independent)	IDs 2/3: Create/Delete Field (functionally independent/dependent field)
1	Change super type	ID 6: Add/Remove a Supertype
1	Change type of a field	ID 7: Generalize Type of Field

time of executing the migrations, e.g. excluding time spent modifying instances in between.

We report the migration time of performing the complete round-trip. We perform each migration for a total of 100 iterations and report the mean execution time. Further, in case a scenario consists of multiple variations (e.g. directions, different modifications), we aggregate the mean across these variants and indicate the corresponding standard deviation.

We report the results of our experiments in the last column of Table 1. The results show that all translation layers migrate the corresponding instances in well under 100ms with a total average of 45ms$\pm$0.023. Further, the use of trace links appears to incur a small overhead as compared to scenarios which do not rely on traceability. Given the architecture of our migration framework, we expect such low execution times, since all migration code is compiled to plain JavaScript which can be executed efficiently with modern JavaScript interpreters. The only overhead incurred by our framework relates to dispatching migration calls at runtime and traceability features.

5.3 Case study: E-commerce Web Application

As the final part of our evaluation, we conduct a practical case study by implementing a concrete translation layer in a realistic setting using our migration framework. We chose a real-world e-commerce web application sourced from an industrial partner. The application implements online shop and reservation systems, the overall development of which comprises more than 10 person years.

The web application uses an API for the communication between web client and server. The data exchanged consists of entities such as orders, products, or reservations. We extracted a data model comprising a self-contained set of 86 classes and 22 enum types. In addition, we restored a three month old version of the data model from the version control system. The current and the restored version formed our representative model versions 1 and 2.

We analyzed the model changes and consulted our catalog of migration scenarios as introduced in Sec. 5.1. From the given model versions, we extracted a total of 12 distinct changes affecting 7 different type declarations, all of which could be related to at least one of the catalog's scenarios. Overall, we identified 6 different types of edit operations as shown in Table 2. Based on the set of observed changes and the exemplary translation layers implemented, we designed an initial migration strategy. In order to handle the 12 data model changes, we implemented 15 custom migration functions. In sum, the migrations consist of 191 lines of code (leaving out empty lines and comments).

In most cases, the changes occurred in an isolated atomic fashion. Thus, it was possible to almost directly reuse exemplary code snippets from the respective scenarios as templates. In some cases, however, multiple changes overlapped (e.g., the type and the name of a field were changed). For those, the implementation of corresponding migration strategies required special attention. As discussed earlier in Sec. 5.1.4, a migration strategy for overlapping model changes cannot directly be inferred from the round-trip scenarios of its atomic constituents. However, the related scenarios provided useful insights which facilitated the design of a suitable strategy for such overlapping model changes. Lastly, types which did not evolve had to be handled as well. For those cases, we adapted our system to provide a generic fallback migration, which directly replicates instances from one model version to the other by copying. Overall, the development of the translation layer took around 1 person week, including testing and debugging.

For some implemented migrations, we leveraged traceability information. According to our own development experience, the use of trace links was comparatively intuitive whereas using modification detection typically entailed more complex behavior. This was due to the fact that in practice, modification detection can have side-effects that are not initially anticipated by the developer. In addition to the support on a methodological level, we greatly benefited from the tool support of our framework. The visualization of round-trip migrations in terms of object graphs emerged as an aid for spotting inconsistencies and tracking trace links.

5.3.1 On Correctness through Automated Testing

To investigate the correctness of the implemented translation layer with respect to the idea of successful round-trip migrations (cf. Sec. 2.1), we used it to execute a large number of distinct migrations ($\sim$400,000 migrations). We leveraged a random instance generator to obtain a large amount of random test data that exhibited an appropriate degree of diversity (e.g., graph size, cycles, `null`-values, etc.). The generator was configured to generate large object graphs, with an average of 32 objects per graph and a maximum of 1600 objects in one case. Using automated testing, we ran a large number of migrations with the general test assertion that we must never observe any loss of information, i.e., round-trip migrated instances are equal to the original instances.

For round-trip migrations with modification, we also implemented random generators for instance modifications. However, to automate the verification of successful round-trip migrations with modification, we also needed to recreate the corresponding instance modifications in the other model version. Using a modification model, we could implement a generic transformation that bidirectionally translates a modification of one model version to a co-modification of the other version. This allowed us to also check for successful round-trip migrations with modification in a fully automated fashion. Using this method, we again generated random instance data as above, including mutations with an average number of 9 modifications per test input and a maximum of $\sim$1200 modifications in one case.

By running a large number of round-trip migrations with our implemented translation layer, we aimed at assuring evidence, yet not formal proof, of robustness and correctness with regard to our notion of successful round-trip migrations (with modification). In this testing phase, we could not observe any loss of information during round-trip migrations. Considering the number of executed migrations, we are therefore confident that the implemented translation layer behaves as desired.

5.4 Discussion and Limitations

We believe to have demonstrated that by virtue of the transformational approach advocated, (i) definition and development of migration functions can be performed and (ii) execution of migrations can be executed timely. Furthermore and from a methodological point of view, (iii) development can be supported by the catalog since it addresses typical scenarios. In the following, we reflect upon design requirements DR1-DR3, and discuss limitations of our approach.

As per DR1, our evaluation shows that our migration framework is effective for a wide range of known object-oriented evolution scenarios. This was the focus of the case study of Sec. 5.3 – we believe this was demonstrated over a real-world migration scenario based on a realistic and comparatively large data model. Furthermore, we employed automated testing to assess correctness of the implemented translation layer, addressing DR2. Besides providing a version-aware IDL, the migration specification language provided makes all relevant information (such as version and traceability constructs) easily accessible to developers. Regarding DR3, the reusable design patterns in the catalog represent typical, frequently occurring data model evolution problems along with their template solutions, that developers may utilize to develop migrations. Finally, by performing micro-benchmarking we demonstrated that migration execution time is low, since we can rely on the performance of modern JavaScript interpreters.

Regarding validity of our results, we identify *construct*, *internal* and *external* threats. Although we could successfully relate all observed model changes in the case study considered to scenarios in the catalog, our discussion of overlapping model changes still applies (Sec. 5.1.3). We especially note that the catalog is not intended to be complete, and future applications may enrich it with further migration scenarios occurring in practice, or in industrial applications. This manifests as a threat to construct validity of our approach. On a functional level, we could validate the correctness of our translation layer through extensive testing. This further confirms the applicability of requirement DR2, although the lack of a formal proof of correctness can be considered as an internal threat to the validity of our results. We also note that the number of observed changes, the time period of change as well as the model size were relatively limited. This means that the results of our case study specifically may not apply to other cases, which is a threat to external validity. Finally, although the quantitative micro-benchmarking performance results obtained demonstrate timely migration executions, aggregate, application-level performance effects may naturally differ. This is a common issue raised in domains where micro-benchmarking is employed to address variability and diversity of cases. We identify application performance profiling as a promising avenue of future work, where the priority would be to assess a wide range of composite applications making use of migrations, in tandem with micro-benchmarks, as employed, e.g., in notable relevant cloud approaches [11], [12].

Performance may become an issue if a lot of versions are to be managed simultaneously. E.g., given 10 versions, this may lead to 9 chained migrations (from 1 to 2, 2 to 3 and so forth). However, in practice, we assume three factors mitigating that problem. First, we assume that the service is run in the latest version (since the change of the service is the very reason for the new version in the first place). That is, only clients (using different API versions) need to be adapted. That means that when a new version is introduced, existing migrations may be rewritten in order to directly translate to the latest version (e.g., 1 to 10, 2 to 10 and so forth). Second, changes usually apply to different parts of a model or API. If changes of the API are disjoint, their respective migrations do not affect each other. So, even if a change leads to a new version, many use cases would be unaffected by that change. Third, the introduction of a transformation layer should be combined with some kind of external version management. That is, eventually some versions may be declared deprecated and unsupported, even though it would be possible by means of transformations layers. This would limit the amount of migrations to be maintained by the API team and reduce the number of possibly chained migrations.

6 Related Work

Different instances of the problem of data model evolution and instance migration have been studied in many different fields. We first discuss parallels to database systems (Sec. 6.1), which tackle very similar problems such as view updating and schema evolution. Next, we discuss web service evolution (Sec. 6.2), which is situated very close to our method due to our JavaScript-based stack and notion of an IDL. Lastly, we review recent work on model-driven engineering (Sec. 6.3), which our framework is based on regarding our notions of migrations and traceability.

6.1 Schema Evolution, Versioning, and View Updates

The problem of synchronizing queries and views upon a schema evolution has been tackled as early as 1982 [13], and appears in a wide variety of application domains under various terms; view/query rewriting, change propagation, mapping adaptations, or co-transformations within the broader research context of bidirectional transformations [14] – see [15] for a comprehensive survey. Essentially, when a database schema is modified, all the code that interacts with the database must be changed accordingly, and co-change analyses can be viable to automate or assist with database application evolution [16]. *Schema evolution* generally refers to the process of facilitating the modification of a database schema without loss of existing data or compromising data integrity [17]. The main aim, however, is to merely update instance data in response to schema changes, which inherently differs from our goal of continuously round-tripping between versions of an API.

The problem within database research is often solved by introducing a data management layer which allows

the schema of a relational database to evolve without the need to rewrite database queries in the application code, e.g., [18]. Data management runs as an abstraction layer at runtime, handling queries and data exchange between the user interface and a database. Recent advances target specific database technologies and focus on runtime evolution [19]. Our data translation layer component follows the same principles, albeit from a model-driven perspective [20] – instead of managing database access, we operate on the data model of the web service, for example exposed through Object Relational Mappers. We treat data representations as data models, and express evolution in terms of models – not database schemas – operating at the level of the Web API.

Schema versioning further implies the ability to access the stored data via arbitrary versions of a schema [21]. One approach of special interest to us is CLOSQL [22]. Here, developers may manually implement so-called update and backdate functions taking care of instance migrations. CLOSQL even implements limited support for mechanisms that we would consider traceability features. However, no methodological support in form of a catalog of re-usable migration patterns is provided to assist developers.

The field of *view update translation* [23] addresses the problem of performing updates on database views which need to be propagated to the underlying instance data or vice versa [24]. Gottlob et al. [25] present the concept of *dynamic views*, which is a database view together with an update policy. This is comparable to our concept of translation layers. In recent work, this idea is further complemented by the use of bidirectional transformation languages from which update policies can be automatically inferred [26], [27], [28]. However, a tacit assumption is that there is always a loss of information from the source to a view, which does not apply in our more general case of data model evolution.

6.2 Web Services Evolution and Specification

Within web services, the problem of interface evolution has been identified as the *chain of adapters* architectural pattern [29], in order to maintain backwards-compatibility with forward changes within a services interface. It concerns keeping a common data store to achieve a consistent state over different versions, with the goal to avoid code duplication by incrementally extending the interface.

A runtime approach where data translation manifests is VRESCo [30], where a versioning mechanism that considers revision management on registry- and client-side is used. Moreover, a general classification of service versioning concepts is presented. VRESCo introduces tags for service versions, giving rise to directed version graphs representing dependencies. Recent advances on web service evolution have also combined concepts from software engineering, e.g., through type based slicing [31] with the objective of allowing multiple versions of a service to be deployed simultaneously while reusing code between versions. We use a similar tagging abstraction as VRESCo – but we focus on transformation of data models of modern Web APIs, instead of transforming code [31]. Moreover, we target JavaScript as a supported language within our framework. This design choice reflects its widespread use in modern web development, quite often characterised by absence of e.g., full-blown WSDL descriptions, where changes can also be extracted [32].

Regarding Web APIs in particular, the OpenAPI Specification and the RESTful API Modeling Language[8] are two popular languages for specifying a Web API by means of an IDL. Frameworks such as Google's Protocol Buffers[9], Apache Thrift or Avro [10] also come with their own IDLs, supporting API versioning and providing annotations (e.g., deprecated) in order to change an API in a backwards compatible way. Avro provides limited support for automatic translation between different versions, such as setting new fields to default values. However, since Avro's IDL does not include object-oriented concepts, multiple scenarios described in our catalog are not supported.

In the literature, the notion of *API evolution* often refers to adding and removing functions or methods in classical libraries, or to change the signatures of their declarations [33], which we explicitly excluded from the scope of this paper. Here, approaches such as [34], [35], [36], [37], [38] have been developed to help client developers in the adaptation to breaking API changes, yet focusing on library method invocations while ignoring the data model evolution problem.

Recently, Seco et al. have also identified the problem of evolving service interfaces and interaction among different data model versions [39]. Based on a formal compatibility model, they automatically generate translation layers for adding/removing/renaming fields. However, their work remains limited to comparatively small edit operations according to their notion of compatibility. In contrast, our framework addresses the more general case of manually implementing translation layers which bridge a much larger gap, including more breaking changes which can only be solved by manually designed migrations (cf. Sec. 2.2).

6.3 Model Co-Evolution and Synchronization

Multiple approaches have been proposed addressing the migration of instance models in response to metamodel changes, referred to as *metamodel evolution and model coevolution* [40]. Most of the work is inspired by research on schema evolution and versioning. The COPE framework developed by Herrmannsdörfer et al. [41] proposes a coupled evolution of metamodels and models using a pre-defined set of operators. Change-based approaches as, e.g., proposed by Cicchetti et al. [42], analyze the metamodel changes and classify them into resolvable and non-resolvable, the latter ones have to be handled by custom migration strategies or interactively during migration. A search-based approach has been proposed by Demuth et al. [43]. The aim is to coevolve models by performing a constraint-based search to restore conformance with the changed metamodel using an off-the-shelf solver. This enables a fully automated migration but, in general, the results of such automated migration or repair are not guaranteed to be the ones desired by the developer [40], [44]. Similar to Herrmannsdörfer [41] and Cicchetti [42], we follow a semi-automated approach in which developers need to define a migration strategy which is then applied during the automated execution of

8. github.com/OAI/OpenAPI-Specification, raml.org
9. developers.google.com/ protocol-buffers
10. thrift.apache.org, avro.apache.org

migrations. We draw on the work of Herrmannsdörfer et al. using their catalog [6] as a basis for our selection of relevant edit operations on object-oriented data models. However, our work differs from all of the aforementioned approaches in two important aspects: Firstly, none of the approaches provides a fine-grained versioning support for metamodels but they rely on external versioning support of a version control system. Secondly, their goal is to merely update instance models in response to metamodel evolution, which inherently differs from our goal of continuously round-tripping between different versions.

Finally, the idea of translation layers can be regarded as a special case of *model synchronization*, which aims at synchronizing models representing different views or levels of abstraction. This is typically supported by dedicated (bidirectional) model transformation systems such as Triple Graph Grammars (TGGs) [45] or the Atlas Transformation Language (ATL) [46]. Bidirectional model transformation cases which are similar to ours have been used as benchmarks at the Transformation Tool Contest [7], [47], yet in a very abstract setting which is disconnected from the practical and technical aspects of the domain of web-based services and their APIs. Incremental synchronizations have been proposed in [48], [49], [50], and Getir et al. [51], [52] have worked on recommender systems that tackle synchronization scenarios where the synchronization strategy is non-canonical. Madari et al. [53] and Getir et al. [52] emphasize the explicit use of trace models between interrelated models to facilitate incremental model synchronization, and the transparent management of trace links being accessible through dedicated language constructs is a common concept in multiple model transformation languages [54], [55], [56]. In contrast to these approaches, which operate on models in the context of model-driven development, our translation layers work on the API level. We do not migrate all the elements (e.g., stored in a database) at once but instead only operate on a subset. That is, we only have to migrate the data exchanged between components, e.g., the parameters used by endpoints. Furthermore, (web) developers tend to be less familiar with declarative transformation approaches such as graph transformation. Since analytical facilities such as confluence or termination analysis of a closed transformation system cannot be transferred to translation layers, we have chosen a more practical approach in which migrations are implemented using a widespread imperative language. However, going forward we also see our work as a foundation for a possibly more declarative migration language on top of our framework.

7 Conclusion and Future Work

In this paper, we have addressed the problem of service API evolution, focusing on the evolution of the common data model shared between system components. Instead of ensuring backward compatibility during data model changes, we proposed the usage of translation layers facilitating the communication of system components relying on different versions of the data model. Fine-grained versioning of type definitions and the imperative specification of migration functions using JavaScript, as provided by our developed migration framework, seem to be a promising approach of implementing such a translation layer. Furthermore, our dedicated migration engine establishes migration modularity on a type-level, which decreases the overall complexity of implementing translation layers significantly.

For all 20 edit operations of a commonly accepted set of edit operations on object-oriented data models taken from the literature, we have been able to define successful round-trip migration strategies. The respective scenarios have been compiled into a catalog and serve as reusable patterns for concrete migration strategies. We were able to validate the approach in a non-trivial case study with an API defining more than 100 types. All the API changes that occurred during a development period of over three months could be related to scenarios of the catalog, and we were able to implement corresponding round-trip migration strategies based on the provided templates. Finally, quantitative micro-benchmarking performance results demonstrated timely execution of migrations.

To date, the task of identifying model changes and writing migrations is left to developers. As for future work, we plan to simplify this task by combining our framework with existing techniques for automatically detecting model changes and generating (initial versions of) suitable migrations. While our framework and IDL show that translation layers are feasible in the context of web-based service APIs, to apply this approach in practice, we need to extend translation layers to also support delegation and adaptation of function calls, and to support more features required by specifications for APIs (e.g., authorization settings or error handling). Moreover, in addition to delegating the migration of referenced objects, we aim at further increasing the level of modularity and reuse by delegating parts of the migration of a subtype to its supertype, e.g., using a dedicated super keyword as known from object-oriented programming. Eventually, we also aim at an empirical evaluation of our domain-specific language. Although we are confident that web developers familiar with JavaScript can deal with the lightweight extensions provided by N4IDL, an assessment of whether the concepts it implements are captured in language constructs would be beneficial.

References

[1] S. Wang, I. Keivanloo, and Y. Zou, "How do developers react to restful api evolution?" in *Service-Oriented Computing*, ser. Lecture Notes in Comp. Sci. Springer, 2014, pp. 245–259.

[2] E. Wittern, "Web apis - challenges, design points, and research opportunities," in *Proc. of the 2Nd Intl. Workshop on API Usage and Evolution*. New York, NY, USA: ACM, 2018, pp. 18–18.

[3] S. Sohan, C. Anslow, and F. Maurer, "A case study of web api evolution," in *IEEE World Congress on Services*, 2015, pp. 245–252.

[4] T. Espinha, A. Zaidman, and H.-G. Gross, "Web api growing pains: Stories from client developers and their code," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, Feb 2014, pp. 84–93.

[5] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, 2013.

[6] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, "An extensive catalog of operators for the coupled evolution of metamodels and models," in *Intl. Conf. on Software Language Engineering*. Springer, 2010, pp. 163–182.

[7] T. B. Anthony Anjorin and B. Westfechtel, "The families to persons case," in *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF)*, 2017.

[8] ECMA TC39-TG1, "Proposed ecmascript 4th edition – language overview," Ecma International, Tech. Rep., Oct. 2007. [Online]. Available: https://www.ecma-international.org/activities/Languages/Language%20overview.pdf

[9] ECMA-262, "ECMAScript 2015 language specification," Ecma International, Standard ECMA-262 6th edition, Jun. 2015. [Online]. Available: http://www.ecma-international.org/ecma-262/6.0/

[10] S. Milton, H. Schmidt *et al.*, "Dynamic dispatch in object-oriented languages," 1994.

[11] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 119–130.

[12] J. Scheuner and P. Leitner, "Estimating cloud application performance based on micro-benchmark profiling," in *2018 IEEE 11th Intl. Conference on Cloud Computing*. IEEE, 2018, pp. 90–97.

[13] B. Shneiderman and G. Thomas, "An architecture for automatic relational database sytem conversion," *ACM Transactions on Database Systems (TODS)*, vol. 7, no. 2, pp. 235–257, 1982.

[14] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2009, pp. 260–283.

[15] L. Caruccio, G. Polese, and G. Tortora, "Synchronization of queries and views upon schema evolutions: A survey," *ACM Transactions on Database Systems (TODS)*, vol. 41, no. 2, pp. 1–41, 2016.

[16] D. Qiu, B. Li, and Z. Su, "An empirical analysis of the co-evolution of schema and code in database applications," in *Proc. of the 9th Joint Mtg. on Foundations of Software Engineering*, 2013, pp. 125–135.

[17] E. Rahm and P. A. Bernstein, "An online bibliography on schema evolution," *ACM Sigmod Record*, vol. 35, no. 4, pp. 30–31, 2006.

[18] A. O. de Bhróithe, F. Heiden, A. Schemmert, D. Phan, L. Hung, J. Freiheit, and F. Fuchs-Kittowski, "A generic approach to schema evolution in live relational databases," in *Intl. Conf. on Information Systems Architecture and Technology*. Springer, 2019, pp. 105–118.

[19] M. de Jong, A. van Deursen, and A. Cleve, "Zero-downtime sql database schema evolution for continuous deployment," in *2017 IEEE/ACM 39th Intl. Conf. on Software Engineering: Software Engineering in Practice Track*, 2017, pp. 143–152.

[20] A. Tolk and S. Y. Diallo, "Model-based data engineering for web services," *IEEE Internet Computing*, vol. 9, no. 4, pp. 65–70, 2005.

[21] J. F. Roddick, "A survey of schema versioning issues for database systems," *Information and Software Technology*, vol. 37, no. 7, pp. 383–393, 1995.

[22] S. Monk and I. Sommerville, "Schema evolution in OODBs using class versioning," *ACM SIGMOD Record*, vol. 22, no. 3, pp. 16–22, 1993.

[23] U. Dayal and P. A. Bernstein, "On the correct translation of update operations on relational views," *ACM Transactions on Database Systems (TODS)*, vol. 7, no. 3, pp. 381–416, 1982.

[24] L. Caruccio, G. Polese, and G. Tortora, "Synchronization of queries and views upon schema evolutions: A survey," *ACM Trans. Database Syst.*, vol. 41, no. 2, pp. 9:1–9:41, May 2016.

[25] G. Gottlob, P. Paolini, and R. Zicari, "Properties and update semantics of consistent views," *ACM Transactions on Database Systems (TODS)*, vol. 13, no. 4, pp. 486–524, 1988.

[26] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem," *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 233–246, 2005.

[27] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi, "Bidirectionalization transformation based on automatic derivation of view complement functions," in *ACM SIGPLAN Notices*, vol. 42/9. ACM, 2007, pp. 47–58.

[28] X. He and Z. Hu, "Putback-based bidirectional model transformations," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 434–444.

[29] P. Kaminski, M. Litoiu, and H. Müller, "A design technique for evolving web services," in *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, 2006, pp. 23–es.

[30] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "End-to-end versioning support for web services," in *IEEE Intl. Conf. on Services Computing*, vol. 1. IEEE, 2008, pp. 59–66.

[31] J. Campinhos, J. C. Seco, and J. Cunha, "Type-safe evolution of web services," in *2017 IEEE/ACM 2nd Intl. Workshop on Variability and Complexity in Software Design*. IEEE, 2017, pp. 20–26.

[32] D. Romano and M. Pinzger, "Analyzing the evolution of web services using fine-grained changes," in *2012 IEEE 19th international conference on web services*. IEEE, 2012, pp. 392–399.

[33] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring: Research articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 83–107, Mar. 2006.

[34] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Trans. on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

[35] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *ACM Sigplan Notices*, vol. 45/10. ACM, 2010, pp. 302–321.

[36] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative apis," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 205–214.

[37] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/xt 0.17. a language and toolset for program transformation," *Science of computer programming*, vol. 72, no. 1-2, pp. 52–70, 2008.

[38] M. Erwig and D. Ren, "An update calculus for expressing type-safe program updates," *Science of Computer Programming*, vol. 67, no. 2-3, pp. 199–222, 2007.

[39] J. C. Seco, P. Ferreira, C. Ferreira, L. Ferrao *et al.*, "Robust contract evolution in a typesafe microservices architecture," *arXiv preprint arXiv:2002.06185*, 2020.

[40] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 396–414, 2017.

[41] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Cope-automating coupled evolution of metamodels and models," in *European Conf. on Object-Oriented Programming*, 2009, pp. 52–76.

[42] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *Enterprise Distributed Object Computing Conf.* IEEE, 2008, pp. 222–231.

[43] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Co-evolution of metamodels and models through consistent change propagation." in *ME@ MoDELS*. Citeseer, 2013, pp. 14–21.

[44] M. Ohrndorf, C. Pietsch, U. Kelter, L. Grunske, and T. Kehrer, "History-based model repair recommendations," *ACM Trans. on Software Engineering and Methodology*, vol. 30, no. 2, pp. 1–46, 2021.

[45] A. Schürr, "Specification of graph translators with triple graph grammars," in *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 1994, pp. 151–163.

[46] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Intl. Conf. on Model Driven Engineering Languages and Systems*, 2005, pp. 128–138.

[47] L. Beurer-Kellner, J. von Pilgrim, and T. Kehrer, "Round-trip migration of object-oriented data model instances," in *Proceedings of the 13th Transformation Tool Contest, part of the Software Technologies: Applications and Foundations (STAF 2020)*, 2020.

[48] G. Bergmann, I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations," *Software & Systems Modeling*, vol. 11, no. 3, pp. 431–461, 2012.

[49] H. Giese and R. Wagner, "Incremental model synchronization with triple graph grammars," in *Proc. of the 9th Intl. Conf. on Model Driven Engineering Languages and Systems*, 2006, pp. 543–557.

[50] M. Wimmer, N. Moreno, and A. Vallecillo, "Viewpoint co-evolution through coarse-grained changes and coupled transformations," in *Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2012, pp. 336–352.

[51] S. Getir, M. Rindt, and T. Kehrer, "A generic framework for analyzing model co-evolution." in *ME@ MoDELS*, 2014, pp. 12–21.

[52] S. Getir, L. Grunske, A. van Hoorn, T. Kehrer, Y. Noller, and M. Tichy, "Supporting semi-automatic co-evolution of architecture and fault tree models," *Journal of Systems and Software*, vol. 142, pp. 115–135, 2018.

[53] I. Madari, L. Angyal, and L. Lengyel, "Incremental model synchronization based on a trace model," in *Proceedings of the 9th WSEAS Intl. Conf. on Simulation, Modelling and Optimization*, pp. 470–475.

[54] F. Jouault, "Loosely coupled traceability for ATL," in *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, vol. 91, 2005, p. 2.

[55] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, "Henshin: A usability-focused framework for emf model transformation development," in *International Conference on Graph Transformation*. Springer, 2017, pp. 196–208.

[56] R. F. Paige, G. K. Olsen, D. Kolovos, S. Zschaler, and C. D. Power, "Building model-driven engineering traceability," in *ECMDA Traceability Workshop (ECMDA-TW)*. Sintef, 2010, p. 49.

Luca Beurer-Kellner is a PhD student at the Department of Computer Science, ETH Zürich, where he is doing research as part of the Secure, Reliable, and Intelligent Systems Lab. Before that he received his MSc in Computer Science from ETH Zrich and his BSc in Computer Science from the Humboldt University of Berlin. His current research interests focus around programming languages, machine learning and networking.

Jens von Pilgrim is a professor for programming methodology at the Hamburg University of Applied Sciences (Germany), and he has more than 15 years of experience in the business world. Before re-entering academia, he worked in a start-up where he lead a team developing a typed JavaScript language (N4JS) and another developing a fully featured online shop. He wrote his PhD thesis at the FernUniversitt in Hagen (Germany) in the Software Engineering group about semi-automated model transformations.

Christos Tsigkanos is a researcher at the Software Engineering Research and Teaching Group at the University of Bern (Switzerland). Formerly, he was Lise Meitner Fellow at TU Vienna (Austria) and post-doctoral researcher at the Distributed Systems Group, as well as at Politecnico di Milano (Italy), where he received (2017) his PhD. His research interests lie in the intersection of distributed systems and software engineering, and include dependable self-adaptive systems and applied formal methods.

Timo Kehrer is a professor at the Institute of Computer Science of the University of Bern (Switzerland), chairing the Software Engineering Research and Teaching Group. Before that, Kehrer was an assistant professor at the Humboldt-Universität zu Berlin (Germany), heading the Model-Driven Software Engineering Group from 2017 to 2021. Kehrer worked as a postdoctoral research fellow in the Dependable Evolvable Pervasive Software Engineering Group at Politecnico di Milano (Italy) from 2015 to 2016, and as a research assistant in the Software Engineering and Database Systems Group of the University of Siegen (Germany) from 2011 to 2015. He has active research interests in various fields of model-driven and model-based software and system engineering, with a particular focus on software and model evolution.