Round-trip migration of object-oriented data model instances

Luca Beurer-Kellner[1], Jens von Pilgrim[2] and Timo Kehrer[3]

[1]ETH Zürich, Zürich, Switzerland
[2]HAW Hamburg, Hamburg, Germany
[3]Humboldt-Universität zu Berlin, Berlin, Germany

Abstract

The communication of web-based services is typically organized through public APIs which rely on a common data model shared among all system components. To accommodate new or changing requirements, a common approach is to plan data model changes in a backward compatible fashion. While this relieves developers from an instant migration of the system components including the data they are operating on, it causes serious maintenance problems and architectural erosion in the long term. We argue that an alternative solution to this problem is to use a translation layer serving as a round-trip migration service which is responsible for the forth-and-back translation of object-oriented data model instances of different versions. However, the development of such a round-trip migration service is not yet properly supported by existing technologies. In this challenge, we focus on the key task of developing the required migration functions, framing this as a model transformation problem.

Keywords

Web development, API and data model evolution, translation layer, round-trip migration, model transformation

1. Introduction

Context: In web development, the communication of web-based services is typically organized through public APIs which rely on a common data model shared among all system components. Over time, the shared data model must be changed to accommodate new or changing requirements, and the system components (i.e., services) including the data they are operating on must be migrated. This API evolution problem is a well-known challenge for web APIs [1, 2, 3].

Figure 1 illustrates this problem by means of a typical example of a distributed system 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, not striped) to version 2 (green, striped), which may lead to different architectural evolution scenarios, depending on the temporal order of updating the involved components. Ideally, all components are updated simultaneously (scenario ❶). When performed in an online fashion, we need a translation layer (TL) to migrate the existing data using tools such as Liquibase[1]. Once the migration has been performed, components relying on version 1 of the data model are replaced by their updated successor versions.

In practice, however, not all the affected components can be migrated instantly and at the same time [4]. A common workaround is to plan data model changes in a backward compatible fashion. However, this severely hampers flexibility when evolving the data model, and essentially comes at the cost of architectural erosion, increased maintenance efforts and technical debt. A more flexible solution would be to operate components relying on different data model versions at the same time and to use a translation layer serving as round-trip migration service being responsible for the forth-and-back translation of object-oriented data model instances of different versions. The evolution scenarios ❷, ❸ and ❹ use such a

[1]https://www.liquibase.org/

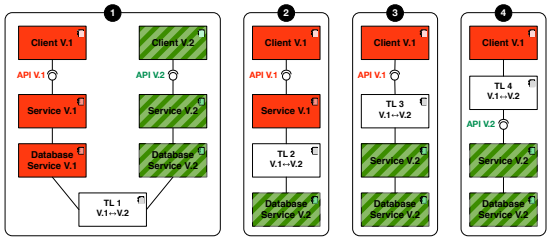

Figure 1: An example of a distributed system. The API and its underlying data model are evolved from version 1 (red, not striped) to version 2 (green, striped).

TTC'20: Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, G. Hinkel, and F. Křikava, 17 July 2020, ~~Bergen, Norway~~ (online).

luca.beurer-kellner@inf.ethz.ch (L. Beurer-Kellner);
Jens.vonPilgrim@haw-hamburg.de (J. v. Pilgrim);
timo.kehrer@informatik.hu-berlin.de (T. Kehrer)
0000-0001-7734-3106 (L. Beurer-Kellner); 0000-0002-7025-8301 (J. v. Pilgrim); 0000-0002-2582-5557 (T. Kehrer)

round-trip migration service to migrate and migrate back shared data model instances on demand. Architecturally, this allows for greater flexibility than the aforementioned solutions. It leaves open a wide variety of design decisions, regarding the use of different data model versions as well as the location of the translation layer (client-side, server-side, in the database system, etc.).

Research Gap: Although it seems to be an attractive solution to deal with data model evolution, the development of a round-trip migration layer which is responsible for the the forth-and-back translation of object-oriented data model instances of different versions is not yet properly supported by existing technologies.

Frameworks such as Google's Protocol Buffers[2], Apache Thrift[3] or Apache Avro[4] support versioning of the whole API and provide annotations in order to change an API in a backwards compatible way. On a more fine-grained level, UpgradeJ [5] extends Java to support versioned type declarations. It allows for upgrading to new versions dynamically at run-time, however, the revised class must have at least the fields and method signatures as the original one. Dmitriev et al. [6] discuss evolution techniques for the PJama persistence framework. Programmers can write migration functions which are embedded by means of static methods. However, there is no dedicated support for implementing round-trip migrations.

Traditional research on data model evolution and instance migration has its roots in the database systems community. Here, *schema evolution* generally refers to the process of facilitating the modification of a database schema without loss of existing data or compromising data integrity [7]. The main aim, however, is to merely update instance data in response to schema changes, which inherently differs from round-trip migrating instances between different versions of an API.

The same limitation applies to more recent work in model-driven engineering. Here, multiple approaches have been proposed addressing the migration of instance models in response to meta-model changes, referred to as *meta-model evolution and model co-evolution* [8]. Their goal, however, similar to schema evolution, is to merely update instance models in response to meta-model evolution. Nonetheless, a multitude of techniques that have been proposed in the context of model evolution and model transformation may serve as a proper basis for the specification of round-trip migrations.

Challenge in a Nutshell: In this challenge, we focus on the key task of developing migration functions which are needed by a round-trip migration service. We only consider API changes affecting the shared data model, while other aspects of API evolution such as signature changes in methods or endpoints in HTTP are out of scope. Protocol changes (e.g., change of message format, authentication, rate limit) as mentioned in Wang et al. [1] are also not considered here. Finally, we focus on a single round-trip migration at a time and do not consider concurrent operations.

We frame the development of migration functions as a transformation problem that abstracts from technological details. While the shared data model is typically defined through Web API specification languages, we choose a more simple and explicit representation using an object-oriented modeling approach. Conceptually, we consider object-oriented data models and instances as graphs, serving as basis for the problem definition which we present more formally in Section 2. Next, in Section 3, we give a set of selected data model evolution scenarios and the corresponding round-trip migration tasks which are to be solved within this challenge. In Section 4, we present criteria for evaluating the submitted solutions. Finally, Section 5 presents a simple reference solution, serving as baseline for more sophisticated solutions based on model transformation concepts and technologies.

An evaluation framework which may be used by solution providers and which comprises a set of experimental subjects is briefly described in Appendix A. The framework as well as a reference solution for this case may be found at https://github.com/lbeurerkellner/ttc2020.

Relation to Previous TTC Cases: At the 2017 edition of the Transformation Tool Contest, the "Families to Persons Case" [9] has been presented. It models a well-known bidirectional transformation problem which is closely related to the underlying problem of our case. However, coming from a more practical setting, we want to emphasize different aspects. As it will become apparent from our evolution scenarios presented in Section 3, our background is mostly motivated by the features of modern web-development languages (e.g., the use of optional fields in Section 3.3) as well as the development process of web applications in general (e.g., our evaluation criterion re-usability in Section 4.4).

2. Problem Definition

In this section, we introduce our conceptual, technology-independent notion of object-oriented data models and instances, and then present properties which we would ideally expect from round-trip migrations.

2.1. Data Models and Instances

Graphs are a natural means to conceptually define object-oriented data models and instances. For the sake of being compatible with the majority of available model transformation technologies, our notion of a graph can be

[2] https://developers.google.com/protocol-buffers
[3] https://thrift.apache.org
[4] https://avro.apache.org

transferred to model representations which are based on the essential MOF (EMOF) standard being defined by the OMG[5]. Specifically, a *graph* $G = (G_N, G_E, src_G, tgt_G)$ consists of two disjoint sets G_N and G_E containing the nodes and the edges of the graph, respectively. Every edge represents a directed connection between two nodes, which are called the source and target nodes of the edge, formally represented by source and target functions $src_G, tgt_G : G_E \rightarrow G_N$. Given two graphs G and H, a pair of functions (f_N, f_E) with $f_N : G_N \rightarrow H_N$ and $f_E : G_E \rightarrow H_E$ forms a graph morphism $f : G \rightarrow H$ if it maps the nodes and edges of G to those of H in a structure-preserving way, i.e., $\forall e \in G_E : f_N(src_G(e)) = src_H(f_E(e)) \wedge f_N(tgt_G(e)) = tgt_H(f_E(e))$.

An object-oriented data model is conceptually considered as a distinguished graph referred to as type graph T, while an instance of this data model is formally treated as an instance graph G typed over T. Formally, a *type graph* $T = (T_N, T_E, src_T, tgt_T, I, A)$ is a special graph whose nodes and edges are representing types, and which comprises the definition of a node type hierarchy $I \subseteq T_N \times T_N$, which must be an acyclic relation, and a set $A \subseteq T_N$ identifying abstract node types. The typing relation between instances and data models may be formalized by a special graph morphism $type_G : G \rightarrow T$ relating an instance graph G with its associated type graph T [10]. The way we handle attributes and attribute declarations follows the definition of attributed graphs given in [11]. The main idea of formalizing node attributes in an instance graph is to consider them as edges of a special kind referring to data values. Analogously, attributes declared by node types of a type graph are represented as special edges referring to data type nodes.

In order to avoid going into any technical details of model transformation approaches yet, we will take an extensional view on data models. That is, speaking about a data model M, then $\mathcal{M}$ refers to the (infinite) set of data model instances which are properly typed over M.

2.2. Round-Trip Migration Functions

We differentiate the *migration* and the *modification* of instances. Given two data models M_1 and M_2 with $M_1 \neq M_2$, a total function $f : \mathcal{M}_1 \rightarrow \mathcal{M}_2$ is considered a *migration function* from M_1 to M_2. Given two instances $m_1 \in \mathcal{M}_1$ and $m_2 \in \mathcal{M}_2$, we say that m_1 is migrated to m_2 if $f(m_1) = m_2$. On the contrary, given a single data model M, a total function $c : \mathcal{M} \rightarrow \mathcal{M}$ is considered an instance *modification function*. Given two instances m and m' typed over M, we say that m is modified to become m' if $c(m) = m'$.

To allow two components which depend on different data models to communicate with each other, a translation layer is responsible for migrating instances forth and back. Formally, a *translation layer* is a tuple $T = (M_1, M_2, f, g)$ where M_1 and M_2 denote the data models the layer translates from and to via migration functions $f : \mathcal{M}_1 \rightarrow \mathcal{M}_2$ and $g : \mathcal{M}_2 \rightarrow \mathcal{M}_1$, respectively. Given an instance $m_1 \in \mathcal{M}_1$, we refer to the consecutive application of f and g to m_1, i.e., $g(f(m_1))$, as the *round-trip migration* of m_1 via M_2. Likewise, since translation layers are supposed to work symmetrically in either direction, given an instance $m_2 \in \mathcal{M}_2$, $f(g(m_2))$ denotes the round-trip migration of m_2 via M_1. The round-trip migration of an instance m_1 via M_2 (resp. m_2 via M_1) is called *successful* if $g(f(m_1)) = m_1$ (resp. $f(g(m_2)) = m_2$). A translation layer T is considered *successfully round-trip-migrating* if the following conditions hold:

$$\forall\, m_1 \in \mathcal{M}_1 : g(f(m_1)) = m_1 \quad (1)$$

$$\forall\, m_2 \in \mathcal{M}_2 : f(g(m_2)) = m_2 \quad (2)$$

In practice, round-trip migrations as introduced above will barely happen since, more often than not, a component will not directly return an instance it just received but rather apply some modification to the instance before returning it. Given two data models M_1 and M_2, a *round-trip migration with modification* of an instance $m_1 \in \mathcal{M}_1$ via M_2 is a consecutive application of functions $g \circ c_2 \circ f(m1) = g(c_2(f(m1)))$ where, like above, f and g are migration functions from M_1 to M_2 and M_2 to M_1, respectively, and $c_2 : \mathcal{M}_2 \rightarrow \mathcal{M}_2$ is an instance modification function performing the modification of the migrated instance $f(m_1) \in \mathcal{M}_2$. Due to the modification of $f(m_1)$, the original definition of a successful round-trip migration is not suitable anymore. The result of migrating back the modified instance $c_2(f(m_1)) \in \mathcal{M}_2$ is not expected to be the original instance m_1. Intuitively, the result is rather expected to be a modification $c_1(m_1)$ of instance m_1 where $c_1 : \mathcal{M}_1 \rightarrow \mathcal{M}_1$ represents the corresponding co-modification of c_2 on data model M_1. A translation layer $T = (M_1, M_2, f, g)$ which handles round-trip migrations between data models M_1 and M_2 is called *successfully round-trip migrating with modification* if there are co-modifications $c_1 : \mathcal{M}_1 \rightarrow \mathcal{M}_1$ and $c_2 : \mathcal{M}_2 \rightarrow \mathcal{M}_2$ such that the following conditions hold:

$$\forall\, m_1 \in \mathcal{M}_1 : g(c_2(f(m_1))) = c_1(m_1) \quad (3)$$

$$\forall\, m_2 \in \mathcal{M}_2 : f(c_1(g(m_2))) = c_2(m_2) \quad (4)$$

3. Selected Evolution Scenarios

In the following sections 3.2 through 3.4, we introduce a selection of different cases of data model evolution and according round-trip migration scenarios. Data models

[5] https://www.omg.org/spec/MOF

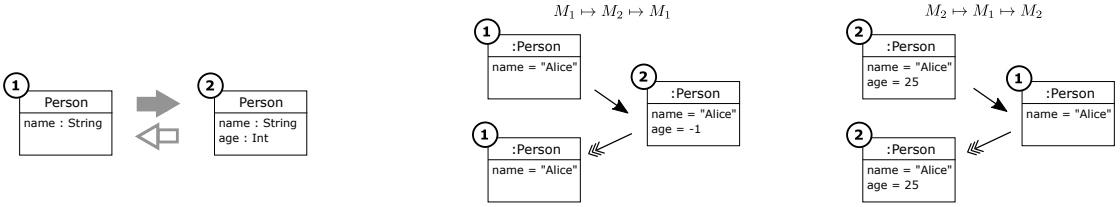

Figure 2: Illustration of the data model evolution scenario "Create/Delete Field" (left) and the corresponding round-trip migrations $M_1 \mapsto M_2 \mapsto M_1$ and $M_2 \mapsto M_1 \mapsto M_2$ (right). Requested specifications for the latter are referred to as Task_1_M_1_M_2_M_1 and Task_1_M_2_M_1_M_2, respectively.

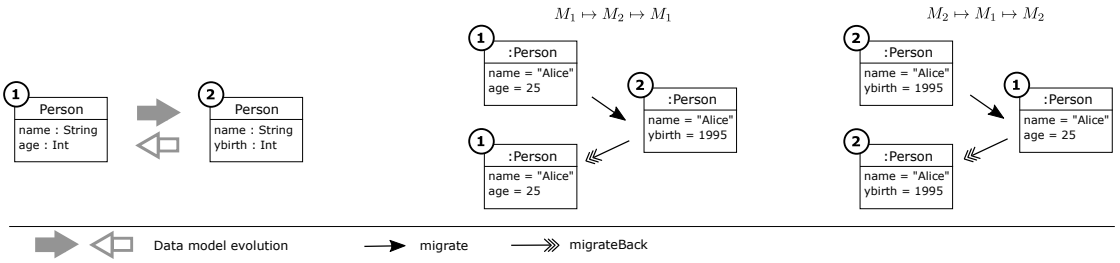

Figure 3: Illustration of the data model evolution scenario "Rename Field" (left) and the corresponding round-trip migrations $M_1 \mapsto M_2 \mapsto M_1$ and $M_2 \mapsto M_1 \mapsto M_2$ (right). Requested specifications for the latter are referred to as Task_2_M_1_M_2_M_1 and Task_2_M_2_M_1_M_2, respectively.

and instances are represented using UML class and object diagram notations, respectively. Each scenario comprises two versions of a data model that demonstrate the application of typical edit operations on object-oriented data models in a minimal context. Each scenario can be interpreted from two perspectives, i.e., from M_1 to M_2, or vice versa. The respective edit operations which can be observed in both cases are inverse to each other. We discuss round-trip migrations in both directions, using the shorthand notations $M_1 \mapsto M_2 \mapsto M_1$ and $M_2 \mapsto M_1 \mapsto M_2$, respectively.

For each of these round-trip migration scenarios, the task is to specify the required migration functions, referred to as *migrate* and *migrate back* in the sequel. That is, each of the four data model evolution scenarios yields two tasks which we ask to be solved by solution providers, summing up to a total number of eight tasks for the entire case. Since all of these tasks are independent from each other, participants may address a subset of them.

3.1. Create/Delete Field

In this scenario, a new field is added to (removed from) a class of the data model, as illustrated in Figure 3 (left). We assume this field to be functionally independent from any other field of the same class.

As illustrated in Figure 3 (right), in a $M_1 \mapsto M_2 \mapsto M_1$ round-trip migration, the new field `age` should be set to some suitable default value since the original `Person` instance does not provide a concrete value for this field. The more complicated case, however, is the $M_2 \mapsto M_1 \mapsto M_2$ round-trip migration since it needs to access a previous revision of the migrated object during a later stage in the round-trip migration. Here, the value of field `age` should be recovered from the original `Person` instance. In the context of traditional bidirectional transformation, this can be considered as a standard scenario which we use as a warm-up task of our round-trip migration case.

3.2. Rename Field

In this evolution scenario, the name of a field is changed. The most simple reason for this kind of change is to improve the wording in the data model to better reflect the terminology of a domain of interest. A more challenging change is to slightly update the meaning of a field, as it is the case in our evolution scenario presented in Figure 2 (left). Here, the field `age` in M_1 is changed to `ybirth` in M_2, now capturing a `Person`'s year of birth instead of its current age.

The migration functions which are to be developed for this scenario should account for this semantic change and convert between proper values of fields `age` and `ybirth`. As illustrated in Figure 2 (right), we assume the current date as a basis for the conversions in both directions. In this case, the change in the semantics of `age` and `ybirth` requires the integration of some user-defined

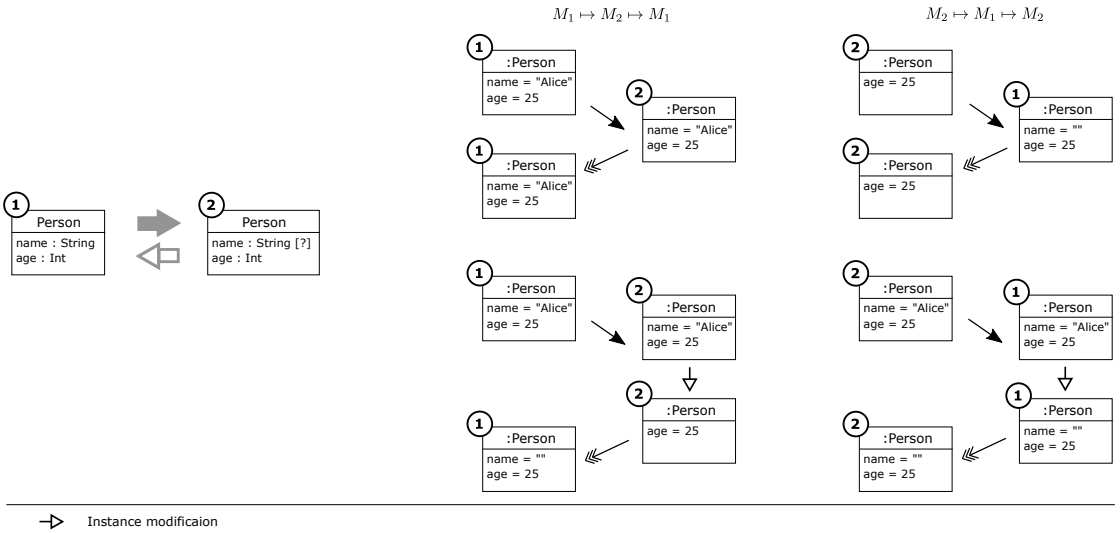

Figure 4: Illustration of the data model evolution scenario "Declare Field Optional/Mandatory" (left) and the corresponding round-trip migrations $M_1 \mapsto M_2 \mapsto M_1$ and $M_2 \mapsto M_1 \mapsto M_2$ (right). Requested specifications for the latter are referred to as Task_3_M_1_M_2_M_1 and Task_3_M_2_M_1_M_2, respectively. The lower example round-trip migration demonstrates how to deal with instance modifications.

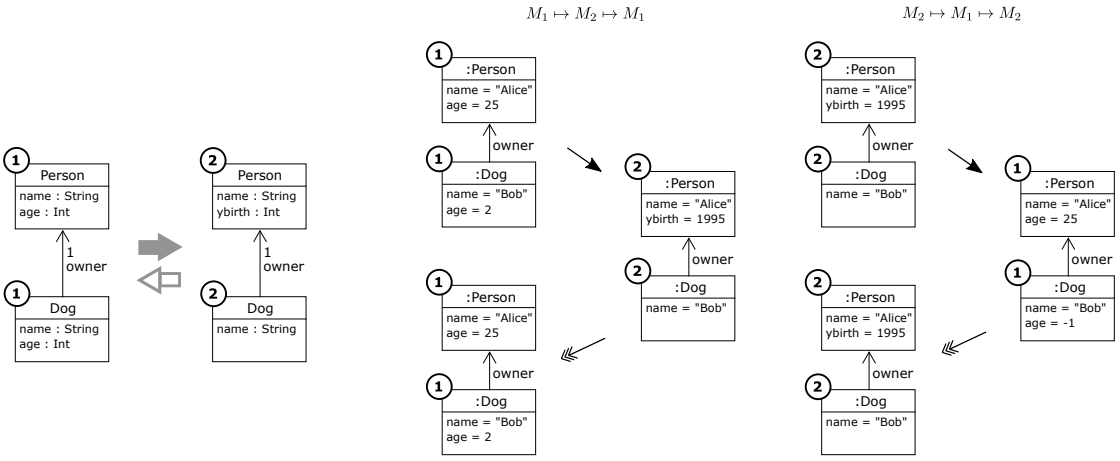

Figure 5: Illustration of the data model evolution scenario "Multiple Edits" (left) and the corresponding round-trip migrations $M_1 \mapsto M_2 \mapsto M_1$ and $M_2 \mapsto M_1 \mapsto M_2$ (right). Requested specifications for the latter are referred to as Task_4_M_1_M_2_M_1 and Task_4_M_2_M_1_M_2, respectively.

arithmetic operation during transformation. Purely structural approaches often lack this feature, even though in our context of Web APIs this is an important requirement.

3.3. Declare Field Optional/Mandatory

In this scenario, the multiplicity of a field is generalized (specialized) from 1 to 0..1 (0..1 to 1). The former case means that the field is declared to be optional, as indicated by the notation `[?]` attached to field `name` in M_2 of the data model shown in Figure 4 (left). The latter case is represented by the default notation used for all other fields, meaning that the field is a mandatory one.

The key issue here is to deal with potential `null`-values in M_2 and their corresponding default values in M_1. This is rather straightforward in a $M_1 \mapsto M_2 \mapsto M_1$ round-trip migration, as illustrated in Figure 4. Here, `null`-values in M_2 may occur due to a modification of the migrated instance, and they should be translated to a default value in M_1. The $M_2 \mapsto M_1 \mapsto M_2$ round-trip

migration is more complicated. Here, we have to check whether a default value has been synthesized during migration or through an explicit modification. In the former case, as illustrated by the upper right example shown in Figure 4, a synthesized default value is migrated back to a `null`-value. In the latter case, illustrated by the lower right example shown in Figure 4, the default value is the result of an explicit modification in M_1, which should be migrated back to a default value instead of a `null`-value in M_2. This evolution scenario is of special interest to us, since optional fields are a common pattern used in the design and evolution of Web APIs.

3.4. Multiple Edits

In this evolution scenario, we combine two edit operations which we have already considered before. As we can see in Figure 5 (left), from an M_1 to M_2 perspective, the field `age` of class `Dog` has been deleted, which corresponds to the edit operation considered in the evolution scenario presented in Section 3.1. At the same time, the name and semantics of field `age` of the referenced class `Person` has been changed to `ybirth`, as in the evolution scenario presented in Section 3.2.

The corresponding $M_1 \mapsto M_2 \mapsto M_1$ and $M_2 \mapsto M_1 \mapsto M_2$ round-trip migrations are illustrated in Figure 5 (right). Their specification can be considered as a combination of the migration functions required for the evolution scenarios presented in Section 3.2 and Section 3.1. The main aim of this scenario is to call for solutions that support some form of re-usability (see Section 4).

4. Evaluation Criteria

To evaluate the quality of the proposed solutions, we give a set of quality characteristics which we consider to be relevant for the specification of round-trip migrations. We draw inspirations from previous work on defining quality attributes of model transformations [12, 13, 14, 15]. We refine each quality characteristic into measurable attributes for each of the tasks presented in Section 3. To obtain concrete measures for their solutions, participants are kindly invited to use the evaluation framework provided with the case resources (see Appendix A). This way, some of the measures can be obtained in a semi-automated manner.

4.1. Expressiveness

A first important and rather obvious quality characteristic is the expressiveness of the transformation language and system being used to specify and execute round-trip migrations. Intuitively, the more data model evolution and according round-trip migration scenarios are supported, the more expressive is the transformation approach.

To turn this intuition into a measurable evaluation criterion, we assess the correctness of each task by providing sets of associated tests. A test case comprises pairs of instances serving as input and as expected output of a round-trip migration. For each of the tasks presented in Section 3, a first test case is derived from the example presented in that section. A second test case is added in order to prevent literal encodings of solutions (except for the taks presented in Section 3.3, which already has two associated test cases. A task is considered to be solved correctly if it passes all tests.

All tasks are scored by means of the provided test cases. A point is given for each passing test case, and points are summarized over all test cases. This means that all tasks are scored evenly between zero and two points. Zero means the task has not been tackled at all, one point indicates a partial solution, and two points mean that the task has been solved and the transformation has been implemented correctly.

4.2. Comprehensibility

Specifications of migration functions should be comprehensible in order to be maintainable and to allow for better manual validation. Our idea of evaluating solutions is to compare their comprehensibility with that of the provided reference solution (see Section 5). For each task, the comprehensibility of the reference solution is scored by one point. Better, equal and worse comprehensibility of a submitted solution are acknowledged by two, one and zero points, respectively.

We acknowledge that such a classification is highly biased by subjective preferences. Developers being familiar with model transformation languages such as Henshin or ATL most likely prefer a declarative or declarative-imperative style, while mainstream web developers will most likely prefer a purely imperative style of writing migrations. More objective measures such as code metrics, as proposed by Götz et al. [16, 17] to compare size and complexity of model transformations written in Java and ATL, are hardly applicable to compare transformations which are written in languages that follow different paradigms (which is to be expected for the different solutions of this case).

To that end, we see two options for assessing the comprehensibility of solutions, both of which involve a human in the loop. In the offline variant, we will use two distinct groups of students to evaluate a solution by answering a survey, similar to [18]. One group of students will have a background on model transformation languages, while the other group is supposed to have only (basic) programming skills (in Java). The second variant is to conduct a live evaluation with the TTC participants.

4.3. Bidirectionality

Bidirectional transformations (BX) [19] appear to be an attractive solution to our problem as they support to synthesize migration functions in both directions from a single specification. Such single specifications may be symmetric as, e.g., in the case of triple graph grammars [20], or asymmetric as, e.g., in the case of putback-based bidirectional programming [21].

Within this challenge, we do not insist on any particular mechanism for specifying bidirectional transformations, and all mechanisms are ranked equally. All tasks and extension tasks are scored evenly with zero (no bidirectionality) or one point (support for bidirectionality).

4.4. Re-usability

As with any other kind of software, re-use mechanisms are an indispensable means to increase the productivity and quality of model transformations. To that end, numerous re-use mechanisms for model transformations have been proposed in the literature, a survey may be found in [22]. We evaluate re-usability by means of the "Multiple Edits" evolution scenario presented in Section 3.4 since it subsumes the scenarios presented in sections 3.2 and 3.1.

One possible option is to achieve re-usability by means of delegation. Specifically, when developing migration functions supporting the round-trip migration of `Dog` instances, this could be achieved by, e.g., delegating the migration of the referenced `Person` instances to migration functions which have been already defined.

Another possible re-use mechanism could be to abstract from the concrete data models and to specify the required migration functions in a generic manner, focusing on the conceptual parts of the respective edit operations. The generic migration functions would then be instantiated for the concrete data model used in this scenario. This is similar to the extraction of core transformation concepts that generalize over several meta-models [23]. In the context of Web APIs, we see this as a core requirement of a feasible transformation approach. In our setting, the continuous evolution of a data model also implies the continuous development of a corresponding migration layer. From a software engineering point of view, a transformation approach should therefore provide support for re-usability. More specifically, s single change to the data model should require only one corresponding change to the migration layer, which implies that existing migration code can be re-used.

We do not insist on any particular re-use mechanism, and all re-use mechanisms are ranked equally. Support for re-usability is acknowledged by four points, while no points are given if the specification has been developed from scratch.

4.5. Performance

Finally, we evaluate the proposed solutions with regards to runtime performance. While the functional correctness of round-trip migrations is an important step towards a valid solution, the Web API context also requires efficient solutions. The implementation of a more complex translation layer would be out of the scope of this challenge. Therefore, as a limited evaluation of the runtime characteristics of the proposed solutions, we repeatedly run the round-trip migrations required to support the evaluation scenarios described in Section 3 for a large number of iterations and measure their execution time. In general however, we consider runtime performance a secondary evaluation criterion. Hence, differences among proposed solutions with regards to runtime performance shall only serve as a tie-breaker among solutions which score equally for the other four criteria.

5. Reference Solution

To provide a reference solution for this case, we implemented all the migration functions which are required to support the 8 round-trip migration tasks arising from our four data model evolution scenarios presented in Section 3 in Java. Its integration into the evaluation framework presented in Appendix A is illustrated in Figure 7 (bottom). Each task is realized by a concrete subclass of class `AbstractTask`, each of which is being instantiated by the concrete task factory called `JavaTaskFactory`. None of the migrations is delegated to a dedicated model transformation system, but the migration functions `migrate` and `migrateBack` are directly implemented in Java.

Qualitative evaluation results Table 1 summarizes the qualitative evaluation results for our Java-based reference solution, namely for the criteria expressiveness, comprehensibility, bidirectionality and re-usability. On the one hand, it is not surprising that a general purpose programming language like Java is expressive enough to

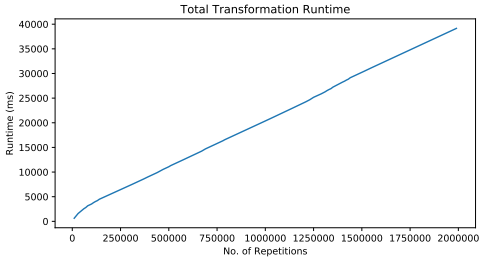

Figure 6: Performance results of our provided reference solution.

Table 1
Evaluation results obtained for the reference solution. Numbers in brackets indicate the maximum score that can be achieved.

Evolution Scenario / Task	Expressiveness	Comprehensibility	Bidirectionality	Re-usability
Create/Delete Field				
Task_1_M_1_M_2_M_1	2 (2)	1 (2)	0 (1)	n.a.
Task_1_M_2_M_1_M_2	2 (2)	1 (2)	0 (1)	n.a.
Rename Field				
Task_2_M_1_M_2_M_1	2 (2)	1 (2)	0 (1)	n.a.
Task_2_M_2_M_1_M_2	2 (2)	1 (2)	0 (1)	n.a.
Declare Field Optional/Mandatory				
Task_3_M_1_M_2_M_1	2 (2)	1 (2)	0 (1)	n.a.
Task_3_M_2_M_1_M_2	2 (2)	1 (2)	0 (1)	n.a.
Multiple Edits				
Task_4_M_1_M_2_M_1	2 (2)	1 (2)	0 (1)	0 (4)
Task_4_M_2_M_1_M_2	2 (2)	1 (2)	0 (1)	0 (4)
	$\sum$: 16 (16)	$\sum$: 8 (16)	$\sum$: 0 (8)	$\sum$: 0 (8)

correctly solve all the tasks provided with this case. Thus, the reference solution achieves the maximum score in this category, i.e., two points per task summarizing to 16 points in total. On the other hand, bidirectionality and re-usability are not supported at all.

Performance results Figure 6 illustrates the runtime characteristics of our reference solution in terms of the performance test of our evaluation framework (see Appendix A). These results were obtained on a Mid-2014 MacBook Pro with an Intel Core i5 processor running at 2,6 GHz and 8 gigabytes of main memory. As expected, the time consumed to perform the round-trip migrations grows linearly with the number of iterations. It takes about 40 seconds to perform all the 2 million iterations of our performance test.

6. Summary and Outlook

In this paper, we outlined our vision of a so-called translation layer which supports the communication of web-based services in different, incompatible versions. One of the key tasks of implementing such a translation layer is to support the round-trip migration of instances of object-oriented data models in different versions. In this challenge description, we phrased this as a model transformation problem which, in contrast to previous TTC cases on the same topic, is driven by the needs and specifics of our application context. We are convinced that modern model transformation technologies such as Henshin [24], VIATRA [25] or ATL [26] are capable of solving the challenge in an elegant way. In particular, solutions to the TTC 2017 "Families to Persons Case" [27, 28, 29, 30] may be adapted to our case with moderate effort.

One of the next steps to further extend this challenge could be to study more evolution scenarios than the four considered in this paper. Moreover, we could think of a (semi-)automated specification of the required round-trip migration functions. Again, we are convinced that technologies from the field of model-driven engineering, notably techniques for model matching [31, 32] and differencing [33], can serve as starting point for such automation.

References

[1] S. Wang, I. Keivanloo, Y. Zou, How do developers react to RESTful API evolution?, in: Intl. Conf. on Service-Oriented Computing, 2014.

[2] E. Wittern, Web APIs - Challenges, Design Points, and Research Opportunities, in: Intl. Workshop on API Usage and Evolution, 2018.

[3] S. Sohan, C. Anslow, F. Maurer, A case study of web API evolution, in: IEEE World Congress on Services, 2015.

[4] T. Espinha, A. Zaidman, H.-G. Gross, Web API growing pains: Stories from client developers and their code, in: Intl. Conf. on Software Maintenance, Reengineering, and Reverse Engineering, 2014.

[5] G. Bierman, M. Parkinson, J. Noble, UpgradeJ: Incremental typechecking for class upgrades, in: European Conference on Object-Oriented Programming, 2008.

[6] M. Dmitriev, M. Atkinson, Evolutionary data conversion in the PJama persistent language, in: Intl. Workshop on Object Oriented Databases, 1999.

[7] E. Rahm, P. A. Bernstein, An online bibliography on schema evolution, ACM Sigmod Record 35 (2006).

[8] R. Hebig, D. E. Khelladi, R. Bendraou, Approaches to co-evolution of metamodels and models: A survey, IEEE Transactions on Software Engineering 43 (2017).

[9] A. Anjorin, T. Buchmann, B. Westfechtel, The families to persons case, in: Proceedings of the 10th Transformation Tool Contest at STAF 2017, 2017.

[10] E. Biermann, C. Ermel, G. Taentzer, Formal foundation of consistent EMF model transformations by algebraic graph transformation, Software & Systems Modeling 11 (2012).

[11] R. Heckel, J. M. Küster, G. Taentzer, Confluence of typed attributed graph transformation systems, in: Intl. Conf. on Graph Transformation, Springer, 2002, pp. 161–176.

[12] E. Syriani, J. Gray, Challenges for addressing quality factors in model transformation, in: Intl. Conf. on Software Testing, Verification and Validation, IEEE, 2012, pp. 929–937.

[13] C. M. Gerpheide, R. R. Schiffelers, A. Serebrenik, A bottom-up quality model for QVTO, in: Int. Conference on the Quality of Information and Communications Technology, IEEE, 2014, pp. 85–94.

[14] K. Lano, K. Maroukian, S. Y. Tehrani, Case study: Fixml to Java, C# and C++, in: TTC@STAF, 2014, pp. 2–6.

[15] S. Getir, D. A. Vu, F. Peverali, D. Strüber, T. Kehrer, State elimination as model transformation problem, in: TTC@STAF, 2017, pp. 65–73.

[16] S. Götz, M. Tichy, T. Kehrer, Dedicated model transformation languages vs. general-purpose languages: A historical perspective on ATL vs. Java., in: MODELSWARD, 2021, pp. 122–135.

[17] S. Höppner, T. Kehrer, M. Tichy, Contrasting dedicated model transformation languages vs. general purpose languages: A historical perspective on ATL vs. Java based on complexity and size, Software and Systems Modeling (2021). To appear.

[18] A. Nugroho, Level of detail in UML models and its impact on model comprehension: A controlled experiment, Information and Software Technology 51 (2009) 1670 – 1685.

[19] S. Hidaka, M. Tisi, J. Cabot, Z. Hu, Feature-based classification of bidirectional transformation approaches, Software & Systems Modeling 15 (2016) 907–928.

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

[21] H.-S. Ko, T. Zan, Z. Hu, Bigul: a formally verified core language for putback-based bidirectional programming, in: SIGPLAN Workshop on Partial Evaluation and Program Manipulation, 2016, pp. 61–72.

[22] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, W. Schwinger, Reuse in model-to-model transformation languages: are we there yet?, Software & Systems Modeling 14 (2015) 537–572.

[23] S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, J.-M. Jézéquel, Reusable model transformations, Software & Systems Modeling 11 (2012) 111–125.

[24] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, M. Tichy, Henshin: A usability-focused framework for EMF model transformation development, in: Intl. Conf. on Graph Transformation, Springer, 2017, pp. 196–208.

[25] D. Varró, A. Balogh, The model transformation language of the VIATRA2 framework, Science of Computer Programming 68 (2007) 214–234.

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

[27] G. Hinkel, An NMF solution to the families to persons case at the TTC 2017, in: TTC@STAF, volume 2026 of *CEUR Workshop Proceedings*, 2017, pp. 35–39.

[28] A. Zündorf, A. Weidt, The sdmlib solution to the TTC 2017 families 2 persons case, in: TTC@STAF, volume 2026 of *CEUR Workshop Proceedings*, 2017, pp. 41–45.

[29] T. Horn, Solving the TTC families to persons case with funnyqt, in: TTC@STAF, volume 2026 of *CEUR Workshop Proceedings*, 2017, pp. 47–51.

[30] L. Samimi-Dehkordi, B. Zamani, S. K. Rahimi, Solving the families to persons case using evl+strace, in: TTC@STAF, volume 2026 of *CEUR Workshop Proceedings*, 2017, pp. 54–62.

[31] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, R. F. Paige, Different models for model matching: An analysis of approaches to support model differencing, in: ICSE Workshop on Comparison and Versioning of Software Models, IEEE, 2009, pp. 1–6.

[32] T. Kehrer, U. Kelter, P. Pietsch, M. Schmidt, Adaptability of model comparison tools, in: Intl. Conf. on Automated Software Engineering, ACM, 2012, pp. 306–309.

[33] T. Kehrer, U. Kelter, G. Taentzer, A rule-based approach to the semantic lifting of model differences in the context of model versioning, in: Intl. Conf. on Automated Software Engineering, IEEE, 2011, pp. 163–172.

[34] C. Brun, A. Pierantonio, Model differences in the eclipse modeling framework, UPGRADE, The European Journal for the Informatics Professional 9 (2008) 29–34.

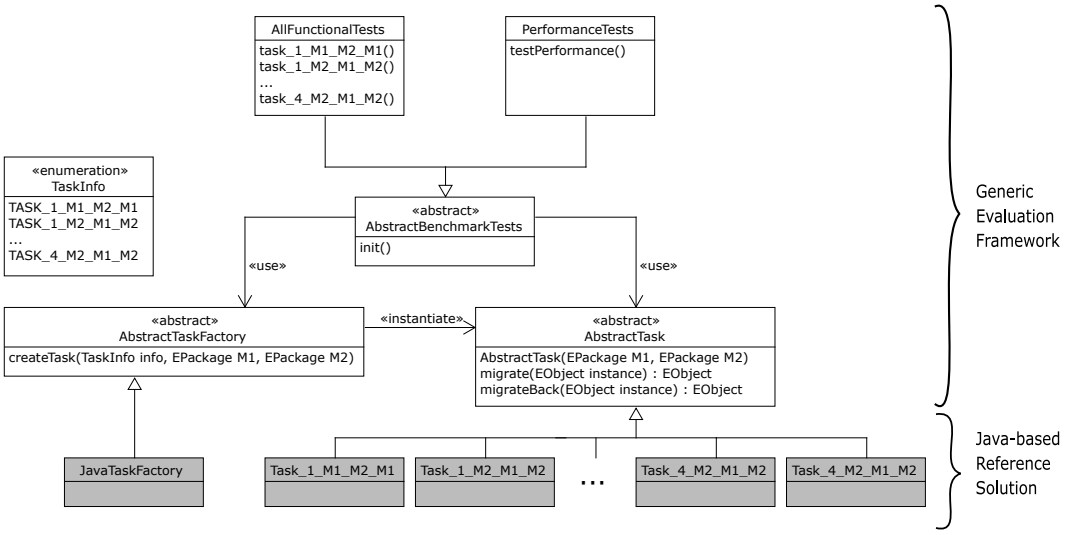

Figure 7: Evaluation framework architecture (top) and integration of the Java-based reference solution (bottom).

A. Evaluation Framework

General architecture Tests in our evaluation framework may be run as JUnit tests. The abstract class `AbstractBenchmarkTests` serves as a base class for all concrete tests (see below), doing some basic initialization. As illustrated by the architectural overview shown in Figure 7, the class `AbstractBenchmarkTests` takes the client role of an implementation of the *Abstract Factory* design pattern, the classes `AbstractTaskFactory` and `AbstractTask` are supposed to encapsulate concrete solutions. That is, for each of the eight tasks presented in Section 3, solution providers who want to use our evaluation framework are asked to provide a concrete subclass of `AbstractTask` which is to be instantiated by a concrete subclass of `AbstractTaskFactory`. The class `AbstractTask` defines the signatures of the two central migration functions called `migrate` and `migrateBack`, respectively. The idea is that `migrate` and `migrateBack` then delegate the actual transformation task to the model transformation system used in a concrete solution.

Functional tests vs. performance tests All test cases for assessing the *correctness* of each of the eight tasks presented in Section 3 may be run as JUnit tests which are collected in the Java class called `AllFunctionalTests`. Each test method, i.e., `task_1_M1_M2_M1()` through `task_4_M2_M1_M2()`, executes a particular task and checks whether for a given input models the obtained output model looks as expected. Checking the equivalence of an actual and expected round-trip migration result is performed using the model comparison tool EMF Compare [34].

A performance test is provided by the class `PerformanceTests`. There is only one test method, called `testPerformance()`, which proceeds as follows: Similarly, to the functional test cases, the test relies on the correct implementation of the `AbstractTaskFactory` and `AbstractTask`. During performance testing, all test cases provided for the four evaluation scenarios are executed repeatedly. That is, a full round-trip migration, involving calls to `migrate` and `migrateBack` is performed. After a certain number of warm-up iterations, this test loop is repeated for a total of 2 million repetitions. The test method measures execution with the increasing number of repetitions and stores the results into the file `results.csv` at the root of the solution's bundle. See the provided code repository of the evaluation framework regarding plotting scripts for the resulting data.

Registration of a concrete task factory In order to register a concrete subclass of `AbstractTaskFactory`, solution providers may use the Eclipse extension point mechanism. Concrete task factories can be registered through a dedicated extension point[6]. Please note that, in this case, the classes `AllFunctionalTests` and `PerformanceTests` need to be run as *JUnit Plug-In Test*. Alternatively, solution providers may subclass `AllFunctionalTests` and `PerformanceTests` which can be then run as a normal *JUnit test*. In this case, the `init` method of these concrete subclasses must take care of instantiating the concrete task factory. Our ref-

[6] `de.hub.mse.ttc2020.benchmark.concretetaskfactory`

erence solution (see Section 5) implements both options for the sake of illustration.

Test data Finally, since many model transformation tools available in the model transformation research community are based on the Eclipse Modeling technology stack, we provide implementations of the data models used in the evolution scenarios presented in Section 3 in EMF Ecore. Consequently, instances serving as test data for assessing the correctness of transformation tasks are represented as EMF instances (often referred to as instance models in the EMF community).