HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Round-Trip Migration von objektorientierten Datenmodell-Instanzen

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Luca Mathias Beurer-Kellner
geboren am: 15.09.1994
geboren in: Stuttgart

Gutachter/innen: Prof. Dr. Timo Kehrer
Dr. Jens von Pilgrim

eingereicht am: ..... verteidigt am: .....

Abstract

In distributed software systems, a shared data model often represents the common denominator between components (e.g. database systems, applications, APIs). Over time, the data model must be changed in order to accommodate for new requirements. This evolution can often not be performed at the same time and completely, which implies that multiple versions of the same data model must be maintained.
In order to further guarantee the successful interoperation between components of different version, it must be ensured that the model differences do not lead to loss of information or misinterpretation. A common solution to this problem is to design data model changes in backward compatible fashion.

In this thesis, we propose an alternative solution to this problem by introducing a translation layer between components of different data model version. This allows for greater flexibility when changing data models, since is does not require backward compatibility. We introduce the term of *successful round-trip migrations*, which describes the lossless forth-and-back translation of data model instances between versions. Based on an object-oriented data modelling language, we present a framework for the execution and implementation of round-trip migrations. During our work, we have identified *traceability*, as known from model-driven engineering, as a core requirements to allow for successful round-trip migrations. We furthermore present a catalogue of so-called round-trip migration scenarios, which discusses various challenges that need to be faced when implementing round-trip migrations.

In order to evaluate our approach, we carried out a case study by implementing a translation layer for a real-world data model. Using our framework and the scenario catalogue, we were successful in implementing a translation layer which guarantees a successful round-trip migration of instances. Based on these results, we were able further verify our framework, our scenario catalogue and the general concept of round-trip migrations. However, overall we have also identified limits to the idea of round-trip migrations, especially with regard to the set of possible data model changes.

We see potential in the idea of round-trip-migrating translation layers, as it allows for non-backward compatible data model changes, while maintaining system interoperability. Future work will show how the applicability of such layers is to be estimated in a real-world production environment.

Zusammenfassung

In verteilten Software-Systemen formen gemeinsame Datenmodelle oft die Grundlage für eine Kommunikation zwischen Komponenten (z.B. Datenbanken, Anwendungen, APIs). Um veränderten Anforderungen gerecht zu werden, müssen solche Datenmodelle mit der Zeit verändert werden. Oft kann diese Änderung jedoch nicht für alle Komponenten zeitgleich und komplett vorgenommen werden. Deshalb können in einem solchen System mehrere Versionen eines Datenmodells zum Einsatz kommen. Um weiterhin die Funktionalität und Zuverlässigkeit des Systems zu garantieren, muss sichergestellt werden, dass die eingesetzten Versionen keine Unterschiede aufweisen, die im Zusammenspiel zu Datenverlust oder Fehlinterpretation führen können. Dieses Problem wird oftmals gelöst, indem Datenmodell-Änderungen eine Rückwärtskompatibilität garantieren.

Diese Arbeit bespricht einen alternativen Lösungsansatz, in dem mittels Übersetzungsschichten (translation layers) zwischen Komponenten verschiedener Datenmodell-Versionen vermittelt wird. Dies erlaubt größere Freiheiten bei der Änderung des Datenmodells, da Rückwärtskompatibilität kein essentielles Kriterium mehr darstellt. Zunächst wird der Begriff einer *erfolgreichen Round-Trip-Migration* besprochen, welcher die verlustfreie Übersetzung einer Datenmodell-Instanz in eine andere Version und wieder zurück beschreibt. Auf Basis einer objektorientierten Datenmodellierungssprache wird ein Framework präsentiert, welches die Ausführung und Implementierung solcher Round-Trip-Migrationen erlaubt. Während dieser Arbeit hat sich dabei insbesondere Traceability, wie aus der modellgetriebenen Softwareentwicklung bekannt, als essentielles Hilfsmittel erwiesen. Zusätzlich umfasst diese Arbeit einen Katalog sogenannter Round-Trip-Migrations Szenarien, welcher eine Reihe an Problemen bespricht die bei der Implementierung von Round-Trip-Migrationen gelöst werden müssen.

Mit einer Fallstudie wurde das Konzept von Round-Trip-Migrationen weiter ausgewertet, indem für ein produktiv eingesetztes Datenmodell eine Übersetzungsschicht implementiert wurde. Dabei konnte mit Hilfe des Frameworks und Katalogs eine Übersetzungsschicht realisiert werden, welche die erfolgreiche Round-Trip-Migration von Datenmodell-Instanzen erlaubt. Diese Ergebnisse ermöglichten eine initiale Verifikation des Frameworks, des Katalogs und dem Konzept von Round-Trip-Migrationen im Allgemeinen. Insgesamt werden jedoch auch Grenzen der Idee von Round-Trip-Migrationen deutlich, insbesondere bezüglich der Menge an möglichen Datenmodell-Änderungen.

Diese Arbeit gibt einen Ausblick auf das Potenzial von Übersetzungsschichten und Round-Trip-Migrationen, welche eine Änderung des Datenmodells erlauben, ohne dabei Rückwärtskompatibilität zu berücksichtigen. In weiteren Arbeiten bleibt es zu ermitteln, wie die Anwendbarkeit von solchen Übersetzungsschichten in Produktiv-Systemen zu bewerten ist.

Contents

1 Introduction

In large software systems, a data model often serves as a basis for the communication between components. With such a model, developers aim to represent domain entities and relations. Based on it, other software artifacts may be produced such as Application Programming Interfaces (APIs) and database schemas. By using the same underlying data model, all software components are assured to operate on a compatible representation of information and their interoperation is facilitated.

Due to changing system requirements during operation and development, data models are subject to change and therefore undergo evolution. As a consequence, existing software components as well as existing data model instances must be adapted to accommodate for model changes. In most cases however, a complete and simultaneous upgrade of all concerned artifacts is not feasible. For instance, in distributed software systems it is often not possible to simultaneously update all participants without compromising system availability. Sometimes, the first party which issues a data model update may not even control some (third party) components that rely on the previous data model version (cf. Figure 1.1).

To mitigate the problem of data model updates, the models are often developed in backward compatible fashion. Over time, this legacy support can cause redundant model structures and decrease the maintainability of a data model significantly. As a consequence, further development and usage of the model becomes more difficult.

As an alternative to such backward compatible model evolution, this thesis proposes an approach which is based on data model instance migration. Rather than requiring the entire system to operate on the same version of a data model, migration layers translate instances between different data model versions. All software components operate on their version of the data model and all inter-component communication is translated accordingly. This approach allows for all software components to remain unchanged and/or be upgraded gradually. Figure 1.1 illustrates an example of such a version-heterogeneous system with translation layers.

Due to the bilateral relationship of most components, the above-mentioned migration layers must be able to translate instances forth and back from and to any data model version in use. In order to not compromise the system functionality, such round-trip migrations (RTMs) must guarantee data consistency which, in particular, implies the prevention of information loss throughout migrations.

This thesis approaches the problem of round-trip instance migrations in the context of object-oriented data models. We present a framework to support the execution and implementation of round-trip migrations. As a core feature we implement traceability, as known from model-driven engineering [33], on the data model instance level. In many cases, this allows us to realize round-trip migrations without loss of information despite the data model differences being comparatively destructive (e.g. the deletion of features).

In section 2, we formally define our notion of a round-trip migration and give a definition of a *successful* RTM. In section 3 and 4, we introduce an algorithm and framework for the execution and implementation of round-trip migrations. We based the general design of the framework on a systematic review of common data model changes in the context of RTMs. The results of this review are documented in section 5 in the form of a catalogue of 21 distinct round-trip migration scenarios. Section 7 puts this thesis in the context of related work. Lastly, we evaluate the effectiveness of the algorithm and the completeness of the scenario catalogue using a case study, which is based on the change history of a real-world data model. Section 8 concludes this thesis.

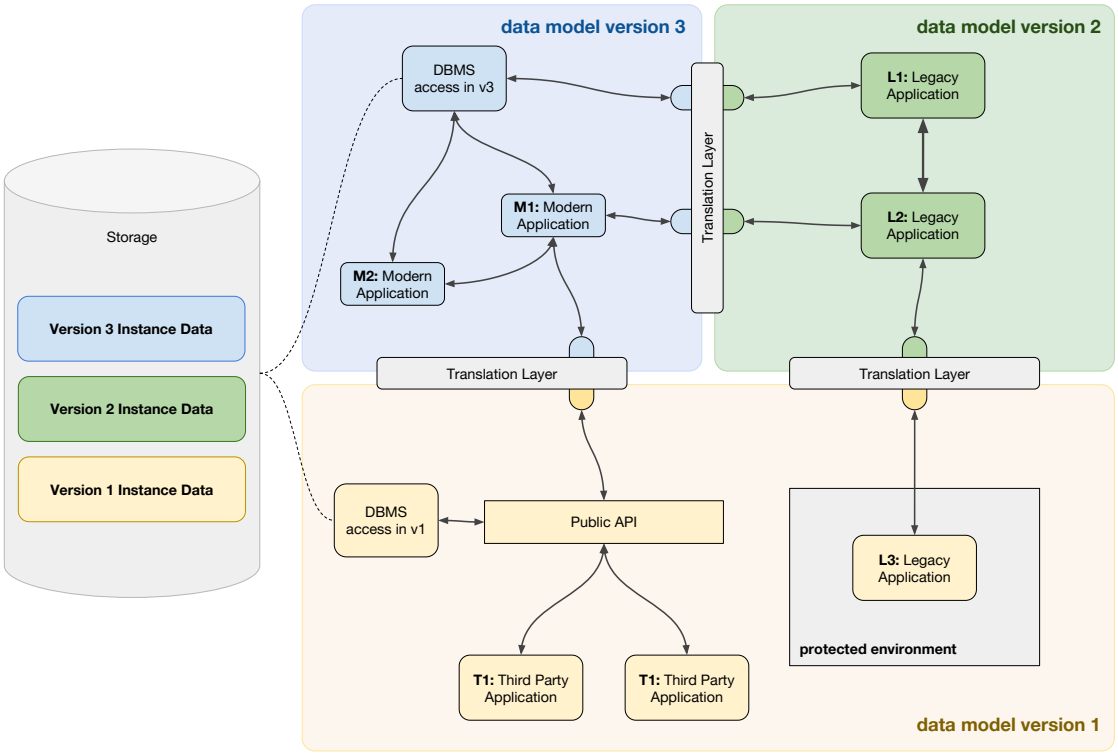


Figure 1.1: An example of a version-heterogeneous distributed system. *Modern Applications* that use the current version of a data model (version 3) reside next to *Legacy Applications* that use older versions of the model. Translation layers ensure a successful communication between components across version boundaries. In this example, persisted data may exist in all available model versions (cf. Storage) and can be translated to the required model version depending on the client version (cf. DBMS access). Using translation layers, modern applications may be linked against the current data model version while older components and APIs (cf. Public API) can still be integrated. Examples of legacy components in this illustration include first party applications which have not been updated yet (cf. L1, L2), third party applications which can only be updated by their vendors (cf. T1, T2) and legacy components which cannot be updated since they are part of a protected environment (e.g. certified hardware/software, hardware implementations, etc.).

2 Foundations and Problem Statement

2.1 Terminology

In order to define the problem of round-trip migrations formally, we first must establish a precise understanding of the term *data model* and *data model instance*. To avoid implementation-dependent details at this point, this section considers a graph-based representation of data models. That is, both on the instance as well as on the model level we think of data models and instances in terms of graphs.

We rely on the concept of type graphs and (typed) instance graphs as described in [25]. A data model is represented in terms of a type graph. An instance graph can then be typed over a type graph using a typing morphism ([25] chapter 3). This is analogous with the common *instanceof* operator in many object-oriented programming languages and formalisms. The nodes and edges of a type graph may additionally be annotated with special markers which allow to model common object-oriented concepts such as inheritance, multiplicity or containment relationships (see [25] for details). An example of a typing morphism is illustrated in Figure 2.1a.

Using this terminology, we now define the *semantics* of a data model as follows:

Definition 2.1. *Data Model Semantics* The *semantics* S_M of a data model M is the set of all conceivable instance graphs which are typed over M in terms of a typing morphism.

This set of conceivable instance graphs can further be seen as one large instance graph that is typed over M. In the following, we will focus on connected sub-graphs in this large graph (cf. Figure 2.1b). We will regard these connected sub-graphs as our notion of *data model instances*. They represent the domain and range for round-trip migrations.

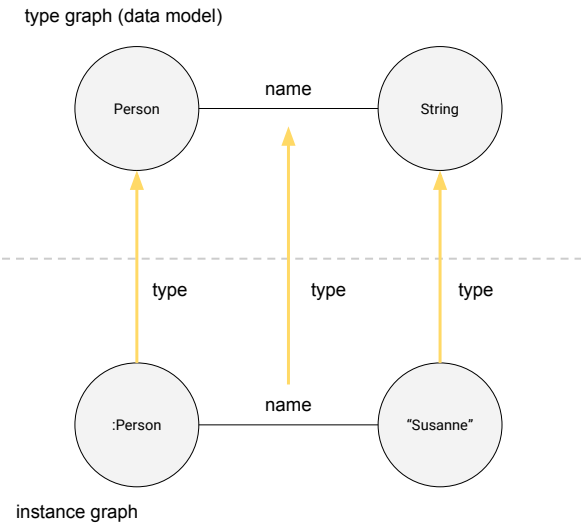


Figure 2.1(a): An illustration of a typing morphism $type = \{$ (:Person, Person), (name, name), ("Susanne", String) $\}$

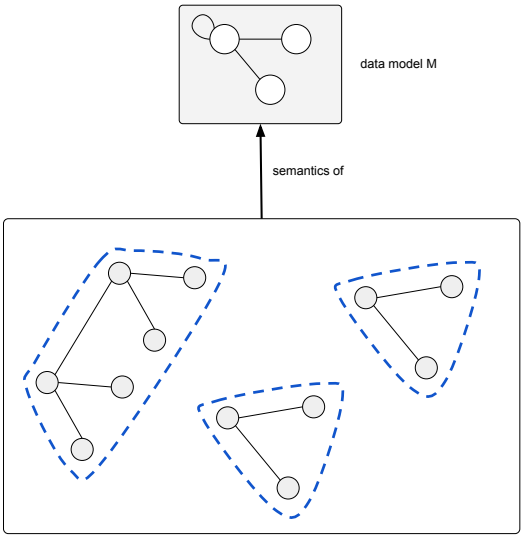


Figure 2.1(b): The semantics of a data model M. The dashed line marks the connected sub-graphs in the semantics of M. Since the semantics of M is possibly infinitely large, this illustration only depicts representative parts of the instance graph.

Definition 2.2. *Data Model Instance*

- Let M be a data model with semantics S_M.
- Let $connected(S_M)$ denote the set of connected sub-graphs of given semantics S_M.

We define the set of data model instances of M as:

$$I_M := connected(S_M)$$

A single data model instance is thus a connected sub-graph in the semantics S_m of M.

Notation: Given two data models M_1 and M_2, a function $f : M_1 \mapsto M_2$ denotes a function with domain I_{M_1} and range I_{M_2}.

It follows that the semantics of a data model M may alternatively be seen as the set of conceivable data model instances of M. Observe that every data model instance of M is an instance graph that can be typed over M. On the other hand, not all instance graphs that can be typed over M are data model instances of M.

This notion of a data model instance allows us to consider connected sub-graphs in the set of all conceivable instances. We assume this perspective as we expect the different peers of a version-heterogeneous system to exchange instance data of non-trivial structure. In particular, we expect data model instances that represent multiple domain entities of strong and weak nature which in turn reference each other in arbitrary fashion (e.g. circular references).

Using these definitions, we differentiate the *migration* and the *modification* of instances:

Definition 2.3. *Instance Migration* Given two data models M_1 and M_2, a total function $f : M_1 \mapsto M_2$ is considered a migration from M_1 to M_2.

A migration can thus be seen as a mapping of the semantics of one data model (version) to the semantics of another data model (version). In contrast, we define an *instance modification* as follows:

Definition 2.4. *Instance Modification* Given a data model M_1, a total function $f : M_1 \mapsto M_1$ is considered an instance modification.

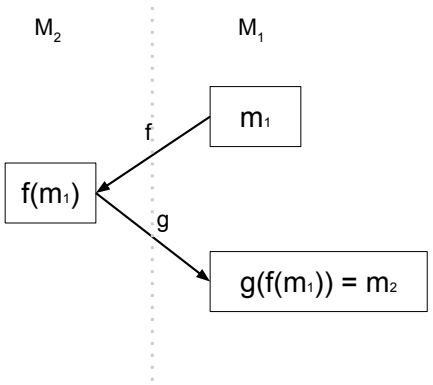


Figure 2.2: A basic round-trip migration of an instance $m_1 \in M_1$ via data model M_2.

2.2 Round-Trip Migrations

To allow two components which depend on different data models to robustly communicate with each other, all messages (in the form of instances) have to be translated to the other model first. Therefore, a translation layer must be provided, which handles the migration of instances. For such a layer to be considered efficient, it must allow for the lossless [1] migration of instances of one data model to the other and back. Furthermore, to enable both components to read and write, it must be possible to translate forth and back from both directions.

Definition 2.5. *Translation Layer* A translation layer is a tuple

$$T = (M_1, M_2, f : M_1 \mapsto M_2, g : M_2 \mapsto M_1)$$

- M_1 and M_2 denote the data models the layer translates from and to.
- f and g denote the migration functions of the translation layer. They map instances of M_1 to M_2 and vice-versa.

Definition 2.6. *Round-Trip Migration* Given a *translation layer* T, we refer to the consecutive application of f and g to an instance $m_1 \in M_1 : g(f(m_1))$ or $m_2 \in M_2 : f(g(m_2))$ as the *round-trip migration* of $m_1 \in M_1$ via M_2 or $m_2 \in M_2$ via M_1 respectively.

An illustration of a round-trip migration is given in Figure 2.2.

Due to their domain and range, the migration functions f and g guarantee the eventual conformance of migrated instances with their original data model. However, based on our initial motivation this is not sufficient. For a ”useful” *translation layer* we furthermore require the preservation of instance identity:

Definition 2.7. *Successful Round-Trip Migrations* The round-trip migration of an instance $m \in M_i$ via another data model M_j is considered to be successful, if it holds true that:

$$g(f(m)) = m \quad [a]$$

A *translation layer* $T = (M_1, M_2, f, g)$ is considered *successfully round-trip-migrating* if it holds true that:

$$(\forall m_1 \in M_1 : g(f(m_1)) = m_1) \wedge (\forall m_2 \in M_2 : f(g(m_2)) = m_2)$$

[a] Assuming $f : M_i \mapsto M_j, g : M_j \mapsto M_i$

In this thesis we focus on how the concept of *successful round-trip migrations* limits the pairs of data models (M_1, M_2) for which there exists a *successfully round-trip-migrating translation layer*. In particular, our work evolves around object-oriented data models. We present a catalogue in which we list a collection of data model changes which can successfully be round-trip migrated. Furthermore, we discuss general limits and challenges in the problem domain of round-trip migrations.

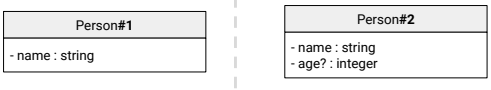
[1] *lossless* as in loss of information.

2.3 Conceptual Limits

Definition 2.7 of *successful round-trip migrations* implies that g is the inverse of f. Therefore, it must be guaranteed that the distinctiveness of two instances m_1 and m_2 is not lost in a round-trip migration. In other words, f and g must not map two instances of one data model to the same instance of the other data model: f and g must be injective.

It is easy to imagine a scenario in which this implication makes it impossible to provide migration functions f and g which fulfil this injectiveness constraint:

Example 2.1. Consider the following data model in version 1 and 2:



(Where the question mark denotes the optionality of field age in version 2.)

In this case, the semantics of model version 2 are a superset of the semantics of model version 1. Version 2 allows for another category of instances which also specify a value for field `age`.

It is clear that a migration function from version 2 to version 1 cannot fulfil the above-mentioned constraint of being an injection. In version 1 there is just no means to express the additional field `age`.

In other cases, such as pure renamings of data model elements, such migration functions clearly exist. Based on these two examples of data model differences, another notion of the round-trip migration problem manifests itself.

Definition 2.8. *Semantically equivalent data models* Two data models M_1 and M_2 are considered to be *semantically equivalent*, if and only if there exists a bijection $b : M_1 \mapsto M_2$.

Any two semantically un-equivalent data models cannot be *successfully round-trip migrated*, since, by definition, a bijective mapping does not exist und therefore at least one of f and g cannot be injective. However, if we consider the two data models to be two revisions in the development of one and the same model, we may assume that most of the time they will differ in more than just semantics-preserving refactorings. A typical data model evolution is more likely to be the active addition or removal of features and thus results in a semantically un-equivalent new model version. Therefore, to overcome the conceptual limits as they were presented in this section, we will focus on additional means of allowing for such model differences (e.g. the inclusion of traceability information). Formally, this means that the domain and range of potential migration functions f and g are extended by additional information in such that the their injectiveness can be guaranteed.

2.4 Instance Modifications

As discussed in the previous section, a *successfully migrating translation layer* is based on two migration functions f and g that invert each other. We discussed the idea of migrating models in both ways and while doing that, preserving the distinctiveness. One important detail however, was the direct application of the function g on the result of function f. In real-world cases, this will happen rarely. A component will not directly return the instance it just received but rather apply a modification to the instance before returning it.

Definition 2.9. *Round-Trip Migration with Modification*

- Let $T = (M_1, M_2, f, g)$ be a translation layer
- Let $c_2 : M_2 \mapsto M_2$ be a function that represents a model modification of an instance $m_2 \in M_2$.

The *round-trip migration with modification* of $m_1 \in M_1$ via M_2 is defined as:

$$(g \circ c_2 \circ f)(m_1) = g(c_2(f(m_1)))$$

For an illustration of a *round-trip migration with modification* see Figure 2.4.

Due to the modification of $m_2 \in M_2$, the original definition of a *successful round-trip migration* (Def. 2.7) is not suitable anymore. Because of the modification of the instance of M_2, the result of a round-trip migration with modification is not expected to be the original value. Intuitively, the result is rather expected to be an equivalently modified instance of M_1:

Definition 2.10. Successful Round-Trip Migration with Modification

- Let $T = (M_1, M_2, f, g)$ be a translation layer where M_1 and M_2 model the same system S.
- Let c be a modification in system S.
- Let $c_1 : M_1 \mapsto M_1$ be a function that represents c in data model M_1.
- Let $c_2 : M_2 \mapsto M_2$ be a function that represents c in data model M_2.

The *round-trip migration with modification* of instance $m \in M_1$ via another data model M_2 is considered to be successful, if it holds true that:

$$g(c_2(f(m))) = c_1(m)$$

The relationship between c_1, c_2 and c is further illustrated in Figure 2.3.

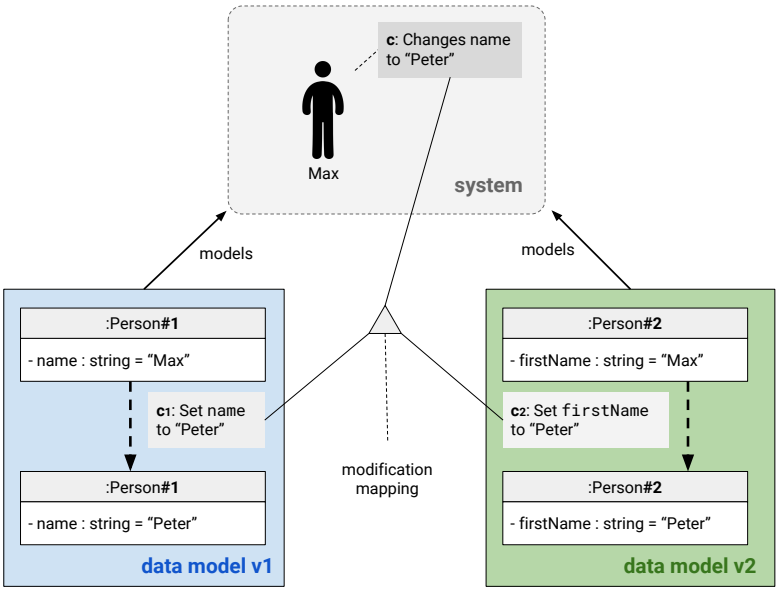


Figure 2.3: The relationship between instance modifications in different versions of a data model (c_1, c_2) and the modeled system (c).

Definition 2.10 demonstrates that the problem of *round-trip migrations with modification* adds another dimension to the initial problem of (simple) *round-trip migrations*. The relationship between c_1 and c_2 imposes a new challenge. The modification c itself does not have to and cannot always be specified formally. However, to allow for this kind of round-trip migration, a translation layer must at least encode some kind of mapping of a modification c_2 to a corresponding modification c_1 in the original data model M_1. Based on how much information on the applied modification is available at migration-time, a concrete implementation may deploy more or less sophisticated strategies to map modifications c_2 back into the original data model M_1.

To conclude this section, we have learned that in order to support *round-trip migrations with modification*, a translation layer does not only have to provide a bijective mapping between the instances of each data model but also between the set of possible operations on the model instances.

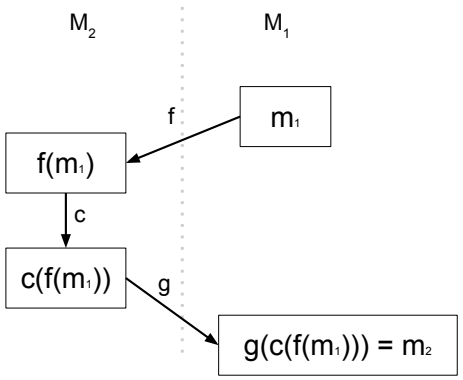


Figure 2.4: A round-trip migration with a modification of the migrated instance of M_2.

3 A Framework for Executing Round-Trip Migrations

While, in the first section, we relied on a graph-based definition of the terms data model and data model instance, we will, in the following, focus on a more practical representation. In the remainder of this thesis, we will make use of the data modelling language N4IDL (NumberFour Interface Definition Language). In the following, we will regard N4IDL data model specifications and corresponding instance data as a concrete example of the initially introduced terms of data models and corresponding instances (cf. section 2). As a consequence, our definitions of (successful) round-trip migrations from section 2 remain valid for the case of a concrete modelling syntax such as N4IDL.

This section serves as an introduction to N4IDL and discusses its syntax and semantics. While the syntax of N4IDL is heavily inspired by the related general-purpose programming language N4JS [1], the semantics of its migrations concept was specified and implemented as part of this thesis. In particular, the scenario catalogue as presented in section 5 was used to identify the core requirements of a framework which allows the implementation and execution of successful round-trip migrations.

3.1 Type Declarations

As an object-oriented data modelling language, the main constructs of N4IDL are the declaration of classes, interfaces and enums:

```
1  class A#1 { // declares a class
2          f : string
3          i : I
4  }
5
6  interface I#1 { // declares an interface
7          s : string
8          a : Array<A>
9
10         optional? : A
11 }
12 enum Coin#1 { // declares an enum
13         HEAD,
14         TAIL
15 }
16
17 // declares a sub-class of A which also implements interface I
18 class SubA#1 extends A implements I {}
```

Snippet 1: Simple type declarations in N4IDL.

Type declarations in N4IDL offer features which are very similar to object-oriented programming languages such as Java. N4IDL is statically typed and supports common concepts such as inheritance and the differentiation between *interfaces* and *classes*. One important syntactic difference to many well-known programming languages is the inverted type notation (`name : string` vs. `string name`).

Classifiers (classes or interfaces) may declare fields of *primitive type* or *reference type*. In the given example, the class A declares the field `f` of primitive type `string`. The field `A.i` is a reference to an instance of type `I` which is declared as an interface (cf. line 6). Fields which are of *reference type* generally express a reference

between objects at runtime.

With regard to multiplicities as known from object-oriented modelling, N4IDL provides constructs for the following cases:

- `0..1` is represented as an optional field (cf. `I.optional` in the given example). Instances with optional fields may or may not hold a value for the given field (e.g. they hold a special `null`-value instead). In different terminology, we refer to a field as *present*, if an instance provides a value for it, and as *absent* otherwise.
- `1` is represented as a simple field (e.g. `A.field`). Such fields are regarded as mandatory and instances are assumed to always provide a valid (non-null) value for them. While this is currently not enforced on a language level, it does make a conceptual difference in the context of round-trip migrations.
- `0..n` is represented as a field of parameterized type `Array` (cf. `I.a`). This includes support for common array operations (e.g. insert, delete, append).

For now, N4IDL does not support any other types of multiplicity on a language level. However, it is possible to model such constraints using custom types (e.g. a custom `Array` type which enforces upper and lower bounds).

3.2 Versioned Types

One important feature of N4IDL is that it allows to declare types in multiple versions. The version of a type is specified together with its name as demonstrated in Snippet 2:

```
1 class B#1 {} // declares type B in version 1
2 class B#2 {} // declares type B in version 2
3
4 class C#1 {
5         b : B // refers to B#1
6 }
7 class C#2 {
8         b : B // refers to B#2
9 }
```

Snippet 2: Versioned types in N4IDL.

In N4IDL, all supported type declarations (classes, interfaces and enums) are considered versionable.

As illustrated in Snippet 2, the version of the currently declared type determines the version of referenced types. For instance, the reference to type B in the declaration of field `C#1.b` will always be bound to version 1 of B. This ensures that, within the data model, it is not possible to declare type references across version boundaries. In N4IDL, the set of type declarations of one version declares a single revision of the specified data model. Therefore, they must not reference each other since each revision of a data model stands on its own.

3.3 Data Model Instances in N4IDL

Initially we defined the subject of migration to be *data model instances* by which we understand connected sub-graphs in a large instance graph, that is typed over a data model (cf. Def. 2.2). This idea applies analogously to our runtime representation of N4IDL, where the subject of migrations are object graphs. These are arbitrarily connected graphs of objects. The root of a migration (*migration root*) is always assumed to be one such object (the initial migration argument), which in turn may refer to other objects that will be migrated recursively. In the following we will therefore refer to the migration input as a whole, that is, the whole object graph which is being migrated, as a *data model instance*. When referring to single nodes in this object graph, we will use the term *object*.

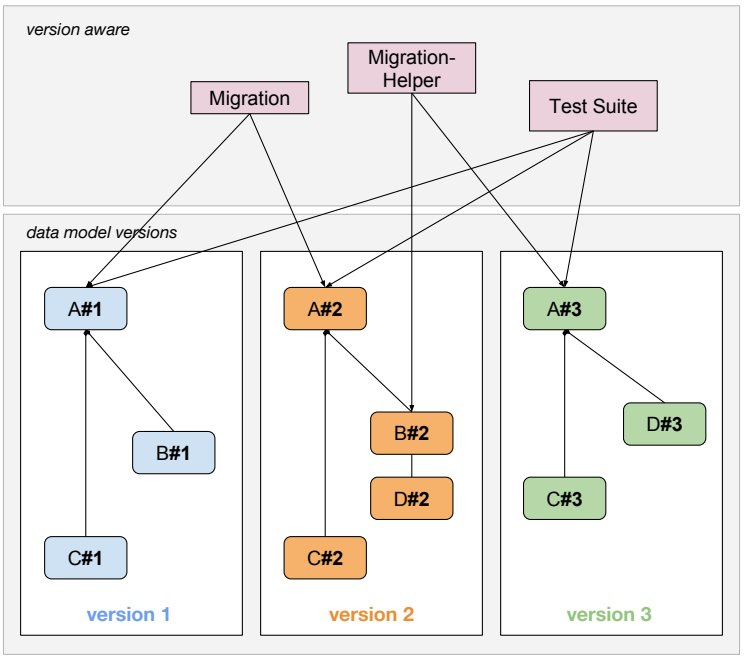


Figure 3.1: Version Aware declarations such as migrations may refer to specific versions of a type, while versioned declarations may only reference types of the same version.

3.4 Version Aware Contexts

Versioned type declarations, as introduced above, can be used to specify the different revisions of a data model. However, N4IDL also provides constructs that allow to write code that deals with and specifies the relationship between different data model versions. In so-called *version-aware contexts*, references to specific versions of a type are allowed. A version-aware context is declared by annotating a class or function as `@VersionAware` (cf. Snippet 3).

Conceptually, version-aware N4IDL declarations do not adhere to the concept of versioned declarations since they usually cannot be assigned to one specific version. Additionally, any reference to version-aware N4IDL code from non-version-aware code (e.g. a versioned type declaration) is prohibited. Examples of version-aware N4IDL declarations are migrations (introduced in the next section), migration helpers or test suites that test version-related behavior. An illustration of how version-aware declarations relate to versioned declarations is given in Figure 3.1.

```
1  class A#1 {}
2  class A#2 {}
3
4  @VersionAware
5  function versionAwareFunction() {
6          var a1 : A#1
7          var a2 : A#2
8  }
9
10 @VersionAware
11 class VersionAwareClass {
12         a : A#1
13         m(p : A#2) {}
14 }
```

Snippet 3: A version-aware function and class that explicitly refer to specific versions of type A.

3.5 Migration Declarations in N4IDL

An important example of version-aware declarations in N4IDL are migrations. An N4IDL migration is a function that migrates between the different versions of a type. Migrations are unidirectional and implemented imperatively. A migration from one version of a type to another is declared on the top-level of an N4IDL file (`.n4idl`) in terms of an annotated function (see Snippet 4)

```
1  class A#1 {
2          f : string
3  }
4  class A#2 {
5          f : string
6  }
7
8  @Migration function migrateA(a1 : A#1) : A#2 {
9          const a2 = new A#2();
10
11         a2.f = a1.f;
12
13         return a2;
14 }
```

Snippet 4: A complete N4IDL module which declares two types and a migration of A#1 to A#2.

In this example, a simple migration of type A in version 1 to type A in version 2 is declared. Since migrations are always version-aware, they can refer to specific versions of a type. To do so, the hash-character may be used in combination with type and constructor references (e.g. line 8 and 9).

Migration declarations may also have multiple parameters and return types:

```
1  @Migration function migrateAB(a : A#1, b : B#1) : ~Object with {a : A#2, b : C#2} {
2          return {
3                  a: new A#2(),
4                  b: new C#2()
5          }
6  }
```

Snippet 5: An N4IDL migration declaration with multiple parameters and return types.

Multiple migration parameters may simply be declared by adding additional parameters to the function declaration. In order to declare more than one return type, one may use the syntax as demonstrated in Snippet 5. On a technical level, the declared migration returns an anonymous object which holds the migration results as its fields.

3.6 Delegation between Migrations via Migration Calls

A very important design goal of N4IDL is modularity. More specifically, it aims at minimizing redundancy in migration code. Therefore, N4IDL encourages to migrate references to other types by delegation. Consider the following type declarations and migrations for type A and B.

```
1  class A#1 {}
2  class A#2 {}
3
4  class B#1 {
5          a : A
6   }
7  class B#2 {
8          renamedA : A
9  }
10
11 @Migration function migrateA(a1 : A#1) : A#2 {
12         return new A#2(); // there is nothing in A to migrate
13 }
14
15 @Migration function migrateB(b1 : B#1) : B#2 {
16         const b2 = new B#2();
17         b2.renamedA = migrate(b1.a); // delegate migration of b1.a
18         return b2;
19 }
```

Snippet 6: `migrateB` delegates the actual migration of `b1.a` to migration `migrateA`.

The migration `migrateB` makes use of the built-in `migrate`-function, which allows to delegate the migration of `b.a` to another migration (in this case `migrateA`). As a consequence, the migration `migrateB` delegates the migration of objects of type A to `migrateA` without declaring an explicit dependency on this specific migration. The invocation of this `migrate`-function is also referred to as a `migrate`-call or *migration call*.

3.6.1 Dynamic Dispatching of Migration Calls

At runtime, `migrate`-calls are dispatched dynamically based on the arguments' runtime types. More specifically, a migrate-call will at runtime always bind to the migration whose parameters come closest to the arguments' types. For an illustration consider the following example:

```
class A#1 {}
class A#2 {}

class SubA#1 extends A {}
class SubA#2 extends A {}

class B#1 {}
class SubB#1 {}

@Migration function migrateA(a : A#1) : A#2 { return new A#2(); }
@Migration function migrateSubA(subA : SubA#1) : SubA#2 { return new SubA#2(); }

@Migration function migrateB(b : B#1) : B#2 { return new B#2(); }
```

Snippet 7: Partly overlapping migration declarations with regard to type B, A and more specific subtype SubA.

In the example of Snippet 7, some exemplary migration argument types will be bound as follows:

Argument Type	Bound Migration	Note
SubA	migrateSubA	migrateA would be a valid fit, but migrateSubA is closer to the actual argument type SubA.
A	migrateA	migrateA is the only migration that fits the argument type.
SubB	migrateB	migrateB does not perfectly match the argument type since it only applies to super-type B. However, none of the other migrations fits the argument better.

To be more precise, we define *type distance* in the context of dynamic migration call dispatching:

Definition 3.1. *Type Distance*

- Let A be a subtype of B
- Let $superClassifiers(A)$ denote the set of N4IDL super-classifiers (directly implemented interfaces and super class) of type A.

The type distance of A to B is defined as:

$$d_t(A,B) := \begin{cases} 1 + min\left\{d(S,B) : \forall\, S \in superClassifiers(A)\right\} & \text{if } A \neq B \\ 0 & \text{if } A = B \end{cases}$$

In case A is not a subtype of B, the type distance is defined as ∞.

Since N4IDL allows to declare multiple migration parameters, we must extend this concept to multiple parameters and arguments:

Definition 3.2. *Migration Distance*

- Let M be an N4IDL migration with parameter types $(p_1, \ldots, p_m)$.
- Let $A = (a_1, \ldots, a_n)$ be a list of migration argument types.

We define the *migration distance* of M to A as follows:

$$d_m(M,A) := \begin{cases} d_t(a_1,p_1) + \cdots + d_t(a_m,p_n) & \text{if } n = m \\ \infty & \text{otherwise} \end{cases}$$

Comparable strategies for method binding can also be found in other programming languages. For instance in Java, the support for method overloading implements a similar concept (cf. Chapter 8.4.9 Overloading in [16]).

Note that this metric of *migration distance* may cause ambiguities during the binding of migrations. An instance of such is illustrated in Snippet 8.

```
class A#1 {}
class A#2 {}

class SubA#1 extends A {}
class SubA#2 extends A {}

@Migration function migrateSubAA(a1 : SubA#1, a2 : A#1) : A#2 { return null; }
@Migration function migrateASubA(a1 : A#1, a2 : SubA#1) : A#2 { return null; }
```

Snippet 8: The two migration declarations `migrateSubAA` and `migrateSubA` fit migration arguments of type `(SubA#1, SubB#1)` equally-well in terms of migration distance.

Based on this definition of type and migration distance, a `migrate`-call always binds to the closest migration in terms of migration distance. This ensures, that the executed migration always considers a maximum of the characteristics of the migrated object in terms of its type. Furthermore, we accommodate for the situation that no migration has been provided for a specific type but instead for one of its supertypes.

In some cases, resorting to a migration which was not declared for the concrete runtime type of an object but rather for one of its (transitive) supertypes may also cause problems. Such cases will usually result in the loss of information, since a supertype migration cannot migrate all fields of the concrete runtime type (but only those of the supertype). While traceability support may allow to compensate for this, this feature may in some situations yield undesired migration results.

The dynamic nature of migration call dispatching imposes another important constraint on the declaration of migrations: In N4IDL, it is not possible to declare more than one migration for the same list of parameter types. Since the binding of migration calls is purely based on the parameter types, this would always entail an ambiguity during migration call binding and must therefore be prevented.

3.6.2 Fulfillment of Migration Calls

While `migrate`-calls may seem like regular function calls, it is important to note, that their runtime semantics differ in one very important aspect. Unlike other function calls, a `migrate`-call may not directly return the corresponding migration result. This entails that it is not possible to rely on the results of a `migrate`-call for any further computations but simple assignments. The only guarantee that is given by a migration call is that the returned value will eventually be replaced with the actual migration result. In that case, we speak of the *eventual fulfillment* of a *migration call*.

The reason for this constraint, is the fact that N4IDL allows to issue `migrate`-calls for parts of the object graph, that are currently being migrated. In other words, the result of a migration call may depend on the result of the currently executed migration. Therefore, the order of executed migrations is not fully controlled by the order in which migration calls are issued. While this imposes a limitation on migrations, it also lowers the overall complexity since migrations can be implemented without minding any cyclic or redundant structures in the migrated object graph.

To summarize, the fulfillment of migration calls are characterized by the following properties:

Definition 3.3. N4IDL Migration Call Fulfillment

The following properties always hold true for the fulfillment of migration calls:

1. A migration call may not directly return the corresponding migration result.
2. All migration calls will eventually be fulfilled.
3. Migration calls with identical migration arguments, result in the same instance of migration result. The corresponding migration function is only executed once.

This minimal set of constraints allows to specify migrations independently from the order in which they are executed. For an example of how the concrete order of execution can be determined, see the description of our reference implementation in section 4.

3.7 Context Information in Migrations

Our initial consideration of the problem of round-trip migrations in section 2 has shown, that in some cases, additional context information is required to implement successful round-trip migrations. In particular in the case

of *semantically non-equivalent models*, context information becomes essential (cf. section 2.3). In N4IDL, it does therefore not suffice for migrations to be simple functions that return objects of another type version, purely based on one or multiple objects of the original version.

To accommodate for this requirement of using context information in migrations, N4IDL introduces the notion of a so-called *migration context*. The migration context manifests itself in terms of a designated `context` object that is available in the scope of a migration declaration. From a developer's perspective, a migration context mainly serves the purpose of providing context information to migrations. More specifically, it allows migrations to access traceability information with regard to the currently migrated instances. Snippet 9 illustrates an example of how a migration may access context information via the `context` object.

On a language level, N4IDL supports three types of context information: (1) access to previous revisions of an instance via trace links and (2) the detection of modifications of migrated instances. These two types of context information allow for traceability which is detailed more thoroughly in section 3.7.1. Additionally, a simple key-value storage for arbitrary values allows for the third type of context information of (3) generic user data.

```
1  class A#1 {
2          field1 : string
3          field2 : string
4  }
5
6  class A#2 {
7          field1 : string
8  }
9
10 @Migration function mA(a2 : A#2) : A#1 {
11         const a1 = new A#1();
12         const previousRevision = context.getTrace(a1)[0] as A#1;
13
14         // Check via the context whether 'a2.field1' has been
15         // modified since it was migrated to the current version.
16         if (context.isModified(a2, 'field1')) {
17                 // special handling for that case
18         }
19
20         a1.field1 = a2.field1;
21         // use the value of field2 of a previous revision of a2
22         // or choose a default value if non-existent.
23         a1.field2 = previousRevision.field2 || "defaultValue";
24
25         return a1;
26 }
```

Snippet 9: An N4IDL migration which accesses the `context` object (migration context). In line 16, the migration checks for a modification of field `field1` since the last migration. In line 12, it obtains the previous revision of `a2` via the captured trace links from previous migrations.

3.7.1 Traceability in N4IDL

N4IDL explicitly integrates two types of traceability. Firstly, it provides one-to-many *trace links* with regard to the currently migrated instance(s). Secondly, it provides migrations with the *ability to detect modifications* of instances after they have been migrated to their current version. During the creation of the scenario catalogue of this thesis (cf. section 5), these two types of traceability were identified as essential for successful round-trip migrations. In the following we will detail the concrete concepts behind these two types of traceability in N4IDL.

Trace Links

The concept of trace links in N4IDL is comparable to their definition in model-driven engineering. N4IDL provides traceability in terms of links between in- and output model instances. With that, support for traceability in N4IDL closely corresponds to the definition of the term *trace* by the Object Management Group:

> "A Trace [...] records a link between a group of objects from the input models and a group of objects in the output models." [13].

More specifically, a trace in N4IDL records a one-to-many relationship between the in- and outputs of executed migrations. For more details on how the implementation of traceability in N4IDL relates to other approaches in the field, please see section 7 on related work.

On a technical level, migrations may access traceability information via the `context`-object. An example of how to obtain the trace of an instance is illustrated in Snippet 9, line 12. An invocation of method `getTrace` on

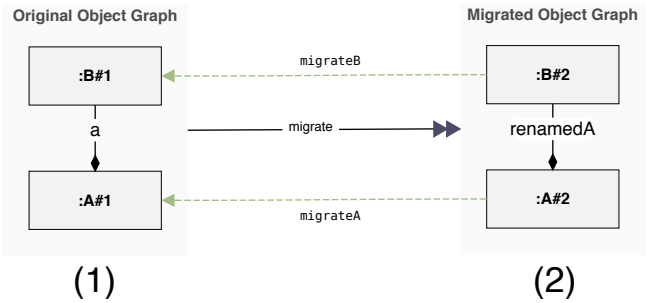


Figure 3.2: The migration of an instance from version (1) to version (2). The light, dashed arrows indicate trace links between the two revisions of the instance.

the `context`-object returns a list of all migration inputs of the previous migration that yielded the given object. In case the given object is not part of a migration result, this list may be empty. On a conceptual level, this corresponds to a one-to-many association. In the following, we will consider the instance(s) and value(s) that are linked to an instance via its trace links, the *previous revision* of that instance. Here, the adjective *previous* refers to the *previous stage* in the preceding chain of migrations.

Figure 3.2 further demonstrates the concept of trace links in N4IDL. The illustration is based on the set of migrations declared in Snippet 6. The instance (1) represents the previous revision of the current revision (2). The light, dashed arrows indicate trace links. As labelled in the illustration, both links can be attributed to the execution of one of the declared migration functions (`migrateB` and `migrateA`). An attribution of trace links to migrations is always possible, since in N4IDL, trace links are created as a side-product of executed migrations.

While the general concept of traceability does, on its own, not involve versions and round-trip migrations, we need to additionally consider another perspective here. In the case of round-trip migrations, we assume that a given instance represents the same system entities in all stages of a round-trip. Therefore, following the trace link of a data model instance, to instances of another version, does not constitute a change of the system entities in consideration. It rather constitutes a change in perspective on the very same set of entities. In migrations, we can exploit this property by using this ability to make up for lost information (e.g. recovering the value of a deleted field by looking at the previous revision).

Above, we assumed that the previous and current revision of a data model instance represent the same system entity. However, this property does not always hold true. In case the instance of the current version has been modified (cf. RTM with modification), this assumption is invalid. The previous revision of an instance now becomes not only the previous revision in terms of the preceding chain of migrations but also in terms of the instance state (the previous revision no longer represents the current state of the instance). Therefore, a migration must always be implemented in accordance with potential functional dependencies and must take the effect of modifications into consideration. Different notions of this problem are discussed in depth in the scenario catalogue in section 5 of this thesis.

Modification Detection

The second type of traceability supported in N4IDL, is the detection of modifications of migrated instances. That is, some migrations may need to incorporate information on whether an instance has been modified since it was migrated to its current version. For an example of such a migration see scenario 5 or 7 in our scenario catalogue. As opposed to trace links, which relate model element to each other, the detection of modifications is supported at the finer granularity of fields. See Snippet 9, line 16 for an example.

The detection of modifications may at first sight not seem like an obvious case of traceability, since modification information does not establish a relationship between elements/instances but rather indicates a fact (of a modification). However, when looking at definitions of the term *traceability* in a broader sense, we may still classify this ability as traceability. Gotel and Finkenstein [17], for instance, define traceability as ”the ability to describe and follow the life of a requirement”. While their perspective on traceability is based in the field of requirements engineering, we will see in section 7 on related work, that there exist parallels to our understanding of traceability.

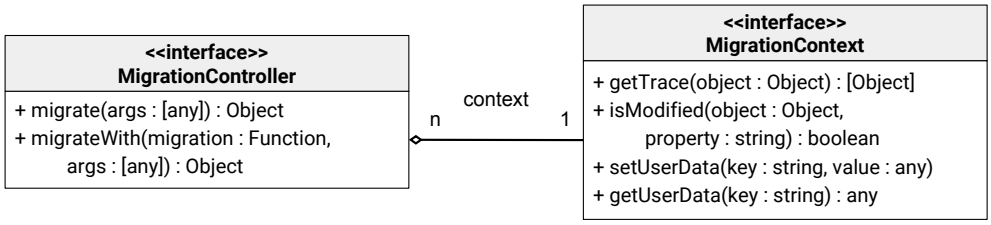


Figure 3.3: The language contract interface of the N4IDL migration runtime.

3.8 The N4IDL Migration Runtime

On the one hand, N4IDL serves as a specification formalism to specify data models and corresponding migrations. However, by the notion of migration calls (cf. section 3.6), N4IDL also specifies a runtime behavior for migrations. To implement this concept of migration calls, N4IDL requires a runtime environment that provides corresponding functionality.

The core responsibilities of the runtime can be structured into three main activities:

1. *Binding Migration Calls* The runtime is responsible for the binding of migration calls given some migration arguments. Based on a list of arguments, it must find the migration (in the set of all declared migrations) that comes closest to the migration arguments' types (cf. Migration Distance, Def. 3.2).
2. *Order of Execution* Given a data model instance, the runtime is responsible for triggering the actual migration functions that are required to fully migrate the instance. This includes determining the order in which migrations are executed. While a basic order of execution is given by how `migrate`-calls are nested, the runtime manages the preservation of object identities and cyclic dependencies.
3. *Capturing and Providing Traceability Information* During the execution of migrations, the migration runtime provides traceability information based on past migrations. Furthermore, it has the role of capturing new traceability information based on the set of executed migrations and their in- and outputs.

Generally, we separate a migration runtime into two main components:

- *Migration Controller* The migration controller is responsible for the binding and execution of migrations. All migration calls are dispatched via the migration controller. Therefore, it determines the order of execution and has the role of populating the migration context with resulting context information (e.g. trace links).
- *Migration Context* The migration context is the central storage for information. Via the N4IDL `context` object (cf. Section 3.7), it is directly exposed to migrations and provides them with traceability information such as trace links and modification detection. It further serves as a global storage for arbitrary user data that may be accessed from all migrations.

The concrete implementation of the runtime environment may vary and can be specified by the program that initially triggers a migration. N4IDL as a language only defines a minimal interface as a contract between language and runtime. In the following this interface will also be referred to as the *N4IDL Language Contract*. The language contract is illustrated in terms of object-oriented interfaces in Figure 3.3.

Given an implementation of an N4IDL migration runtime, or more specifically, of a migration controller, the migration of a data model instance can be triggered as follows:

```
let controller : MigrationController // obtain migration controller instance
let a : A#1 // some object of a type A in version 1

// Use controller to trigger the migration of the object
// graph reachable from a.
let a2 : A#2 = controller.migrate([a]) as A#2;

// After the migration one may access and evaluate the migration context using
let context : MigrationContext = controller.context;
```

Snippet 10: Triggering the migration of an N4IDL data model instance using a migration controller.

In the context of this thesis, we have implemented a reference implementation of an N4IDL migration runtime. Please see section 4 for a description of our implementation..

4 A Reference Implementation of an N4IDL Migration Runtime

In this section we describe our reference implementation of an N4IDL migration runtime. While N4IDL is intended as a modelling language which allows to be compiled to different targets (different programming languages), this thesis is based on an N4IDL generator backend that emits *ECMAScript 5* [10] compliant JavaScript code. We furthermore provide a reference implementation of a migration runtime that works with the output of this backend. This section discusses the inner-workings of our implementation which was specifically designed to provide ideal support for the round-trip migration of data model instances. For instructions on how to inspect the source code of the implementation, please see appendix B.

As mentioned earlier, the core activities of a migration runtime are (1) Binding Migration Calls, determining the (2) Order of Execution and (3) Capturing and Providing Traceability Information. In the following, we will discuss how our implementation approaches each of these points. An overview of the different steps, the reference implementation performs during a migration, is given in Figure 4.1.

4.1 Binding Migrations

In order to accommodate for the requirements of type-distance based migration call binding, our reference implementation leverages the support for reflection in the generated code. The backend emits meta-information on both the model types as well as the migrations as part of its JavaScript output. This information provides the runtime with access to type hierarchies and the set of declared migrations. It allows for the implementation of a type-distance based migration binding strategy in full accordance with the specified binding of migration calls in N4IDL.

4.2 Order of Execution

In contrast to an object graph as migration input, a single N4IDL migration implements only the partial mapping of one object in the overall graph. The migration of the successor nodes (referenced objects) of a node, are triggered by the use of `migrate`-calls. The outputs of these partial migrations must be assembled in a way that ensures the successful migration of the complete object graph (cf. Figure 4.2).

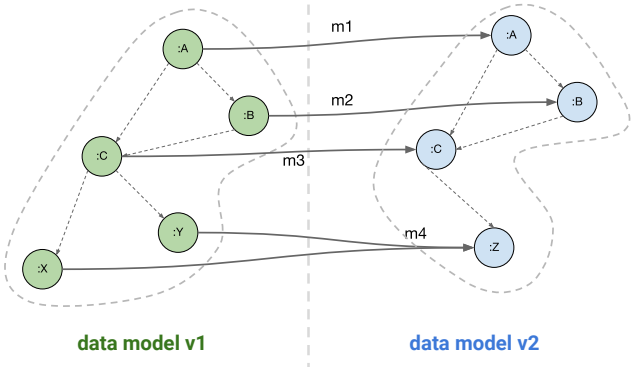


Figure 4.2: A selection of N4IDL migrations (e.g. m1, m2, m3, m4) perform the migration on an object level. For the migration of the whole instance (object graph), the migration runtime must construct the complete migrated instance from these intermediate results.

The nesting of N4IDL `migrate`-calls (cf. Section 3.6) imposes an initial order in which migrations can be executed. However, naively fulfilling each `migrate`-call by simply executing the corresponding migration with the given arguments, will not always yield correct results and does not adhere to the specified properties of migration call fulfillment (cf. Def. 3.6.2, Property 3 in particular). Therefore, in a migration runtime we must deploy a more sophisticated strategy.

For the migration of object graphs we differentiate between different classes of migration inputs. Note that at this point, we assume all input object graphs to be represented as directed graphs. Furthermore, we assume migration-by-traversal (cf. `migrate`-calls) which implies that all migration inputs must be connected graphs. Precisely, we will consider three classes of inputs:

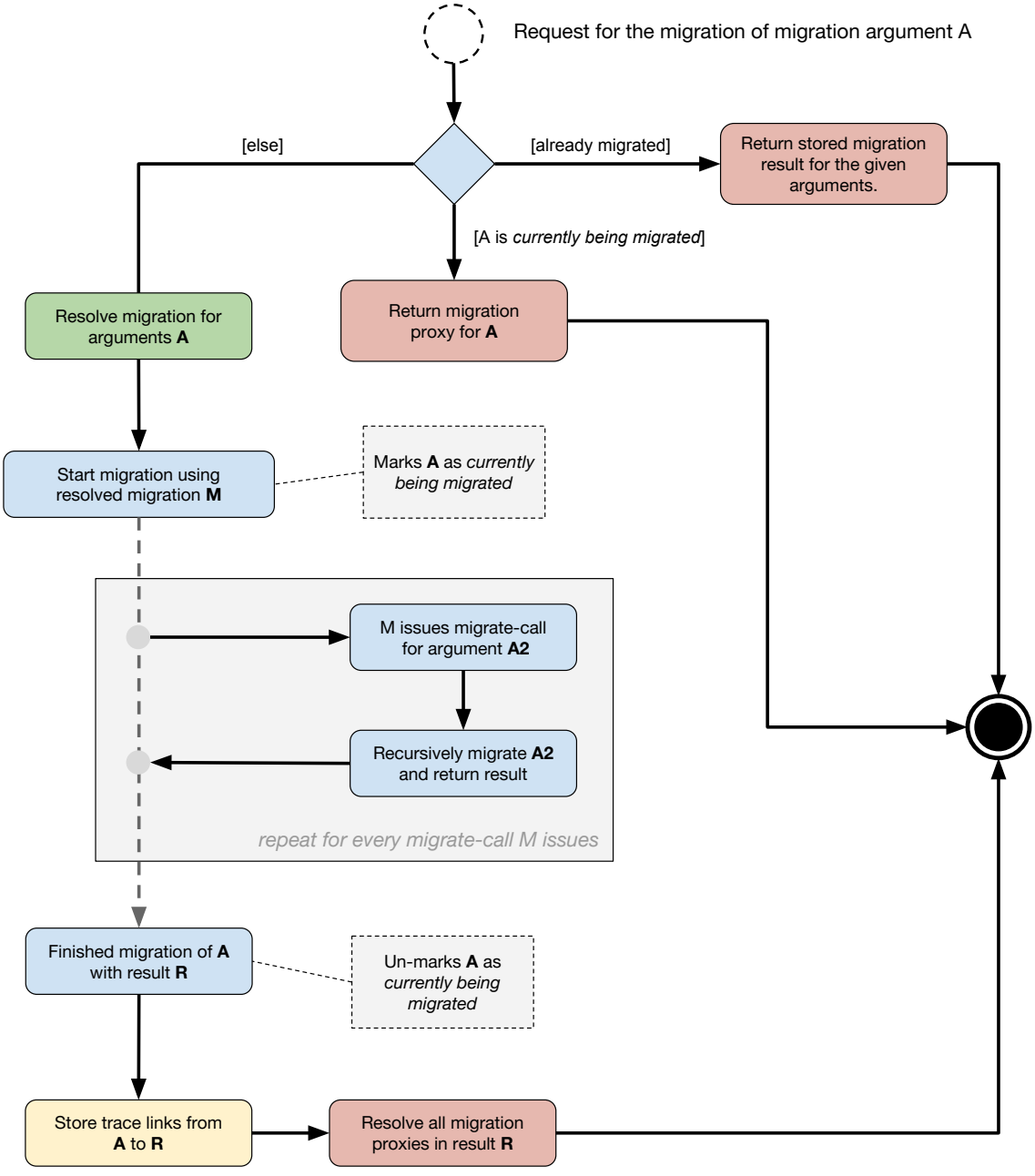
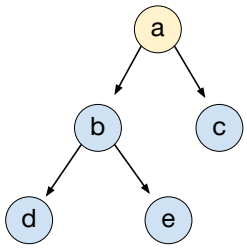
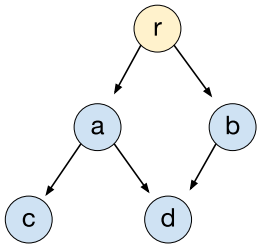


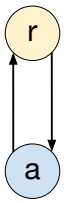
Figure 4.1: The major steps of a migration using our reference implementation of a migration runtime. Green indicates steps related to *Resolving Migrations*, red to *Order of Execution* and yellow to *Capturing and Providing Trace Links*.



1. Object Trees



2. Directed Acyclic Object Graphs



3. Directed Cyclic Object Graphs

1. *Object Trees* The input object graph has the properties of a directed tree. More specifically, the object graph fulfils the constraint that there is exactly one path from the root to any node of the graph.
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. For instance, in the given example, node d can be reached from root r via nodes a or b. Despite its non-tree-like characteristics, this class excludes graphs with cycles and thus maintains some of the advantages of object trees.
3. *Directed Cyclic Object Graphs* The input object graph is a directed graph with cycles. In the given example, the edges $(\{(r, a), (a, r)\})$ form a cycle.

In N4IDL, a migration is expected to issue `migrate`-calls for all the successor nodes that need to be migrated, based on the references found in the currently migrated object. Therefore, references in the input object graph translate to dependencies between migrations. This leads to an important conclusion with regard to these classes of migration inputs:

Object Trees can be migrated by directly fulfilling all `migrate`-calls. Due to the tree-property of the input, no cyclic dependencies can arise and the dependencies of migrations are distinct.

To allow the migration of *directed acyclic object graphs*, we must prevent the duplicate migration of sub-graphs that can be reached by more than one path. Multiple `migrate`-calls with the same inputs must therefore return the same instance of result (cf. Def. 3.3 Migration Call Fulfillment). In our reference implementation, this is achieved by caching intermediate migration results by their arguments.

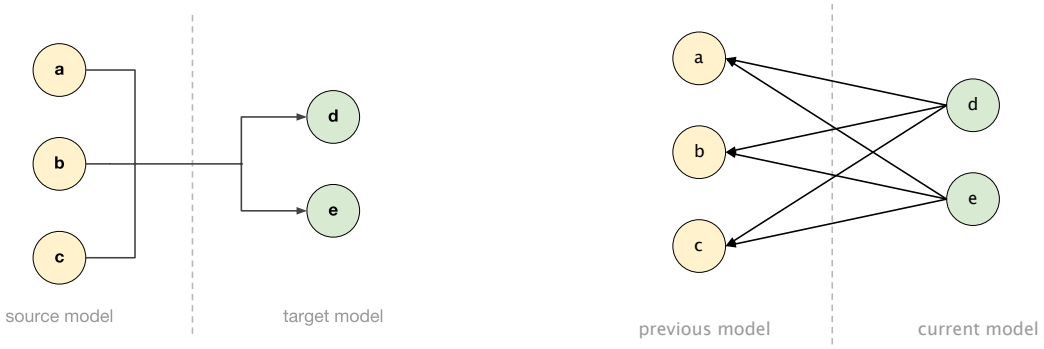
In case of the migration of *cyclic object graphs*, the cycles in the object graphs translate to cyclic dependencies between migrations. More specifically, a migration may potentially depend on its own output. Therefore, we cannot guarantee the immediate fulfillment of `migrate`-calls in cyclic graphs. Our reference implementation detects such cases and returns proxy-objects instead. These represent place-holders which are eventually replaced by the corresponding sub-graph in the migration result. This guarantees the eventual restoration of the original cyclic structures in the migration output. However, it restricts such migrations from further processing the results of cyclic `migrate`-calls. Any operation on proxied migration results fails at runtime, since the actual result is not yet available. To accommodate for this situation, N4IDL as a language, does not guarantee the immediate fulfillment of `migrate`-calls (cf. 3.6.2 Fulfillment of Migration Calls).

Using these strategies for different migration inputs, our reference implementation supports the migration of arbitrarily connected object graphs.

4.3 Capturing and Providing Traceability Information

Since by design, all migrations are dispatched via the migration controller, it has full access to all migration in- and outputs. From this position, it can easily store mappings from instances of the source model (migration arguments) to instances of the target model (migration results). Additionally, N4IDL allows for migrations with multiple in- and outputs. Thus, a migration runtime is required to allow for a many-to-many mapping between instances.

In our implementation, we first capture many-to-many trace links based on the executed set of migrations, arguments and results. We then transform the links to one-to-many links and store them in the migration context. As a consequence, these one-to-many links are exposed to migrations via the `context` object (cf. *Context Information in Migrations*). To achieve this transformation between link cardinalities, we link each migration



(a) An exemplery many-to-many trace link as captured during the execution of migrations in a migration context. Trace links associate source element with target elements.

(b) Two exemplary one-to-many trace links as available from the perspective of a migration (via the `context`-object). Trace links associate current model elements with previous model elements.

Figure 4.4: Different representations of trace links in a migration context.

output with all of the corresponding migration inputs (cf. Figure 4.4a vs. Figure 4.4b).

This strategy allows to re-purpose the set of captured trace links (many-to-many) as cache with regard to the duplicate migration of recurring sub-graphs (cf. directed acyclic object graphs in 4.2 Order of Execution). Furthermore, it defers the transformation of directly inferable many-to-many links to more convenient one-to-many links until the migration to another model version.

To achieve the detection of modifications of migrated instances, we make use of the concept of ECMAScript Proxy Objects (see 26.2 Proxy Objects in [10]). This language construct allows to realize a limited implementation of the well-known *observer* pattern for all migrated instances. With that, any modification of migrated instances can easily be detected and captured as part of the traceability information in our migration context. In a more practical implementation, it may be desirable to achieve the goal of modification detection in a less resource-intensive way. Therefore, it can also be an option to require users of migrated instances to explicitly declare any modification and manually register it with the corresponding migration context.

5 Scenario Catalogue

In this section we present a catalogue of round-trip migration scenarios. The scenarios are discussed in terms of the changes that can be observed on the instance level. To approach the problem systematically, the examples are based on an existing catalogue for object-oriented (meta)model evolution as published by Herrmannsdoerfer et al. [21].

For the catalogue we assume the use of an N4IDL migration runtime as given by the reference implementation of the previous section. On a more technical side, we actually used it to implement the scenario catalogue in code. For instruction on how to inspect the source code of the catalogue, please see appendix C.

Section 5.1 and 5.2 introduce notational aspects and the term of a *round-trip migration scenario*. In 5.3, we present instructions on how to use and understand this catalogue. Section 5.4 forms the main body of the catalogue and discusses the various scenarios in detail. Finally, in section 5.5 we present a selection of our main learning outcomes from the work on the scenarios.

5.1 Notation

In the following we will use a shorthand notation to refer to a round-trip migration. A round-trip migration of a data model instance in version 1 via version 2 is denoted as $\#1 \mapsto \#2 \mapsto \#1$.

Furthermore, we will make use of object-graph diagrams to visualize the effects of a round-trip migration. An example of such a visualization is given in Fig. 5.1. Such round-trip migration diagrams can usually be structured into four consecutive stages: *Original Object Graph*, *Migrated Object Graph*, *Modified Migrated Object Graph* and *Round Trip Object Graph*. They correspond to the (up to) 4 stages of a round-trip migration (with modification) (cf. Def. 2.6 and 2.9). The stages are illustrated in the form of lighter shaded rectangles with corresponding labels. They contain object graphs which depict the data model instances between the application of the different migration and modification functions. In case of a round-trip scenario without modification, the stage *Modified Migrated Object Graph* is omitted.

The black, solid arrows relate the different stages to the application of migrations. The dashed, black arrows indicate a modification from one object graph to the other. The light-green, dashed arrows indicate the trace links that were constructed based on the executed set of migrations and their in- and outputs.

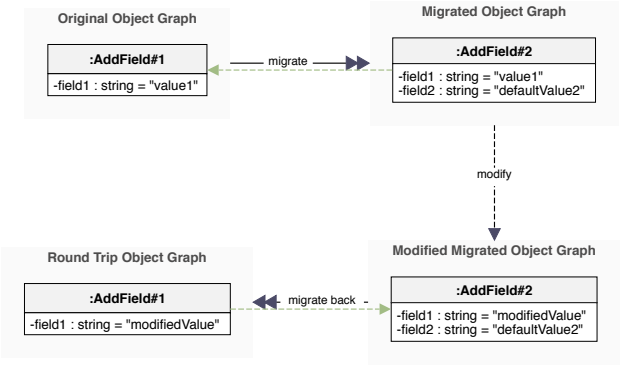


Figure 5.1: An example of a round-trip migration visualization.

Shortened migration listings

In some scenarios, the listings to illustrate the exemplary implementation of migrations is shortened for the sake of brevity. In these cases, the listings were reduced to demonstrate only the core ideas of a migration strategy. However, the full implementation can be found in the source code representation of the scenario catalogue. All shortened listings are marked with an appropriate note.

5.2 Round-Trip Migration Scenarios

A round-trip migration scenario is defined by the following parameters:

- A data model difference D (e.g. *"add a field f to class C"*).
- Two versions of an exemplary data model M_1 and M_2 which differ in difference D.
- Implementations of migration functions f and g (cf. Def. 2.5).
- An example instance $m_1 \in M_1$.

In many cases we will also consider a modification of the migrated instance (cf. section 2.4). In these cases, a scenario is additionally specified by:

- A modification c_1 of the migrated instance $m_2 \in M_2$.
- A modification c_2 of the original instance $m_1 \in M_1$ which represents c_1 in terms of M_1 (cf. Def. 2.10).

Given a model difference D, we may construct corresponding M_1 and M_2 as well as f and g. However, other parameters such as m_1, c_1 and c_2 need to be varied to cover all possible scenarios that stem from D. Furthermore, for each scenario we must consider both directions, resulting in the discussion of RTMs $M_1 \mapsto M_2 \mapsto M_1$ and $M_2 \mapsto M_1 \mapsto M_2$.

In this catalogue, the general goal is to provide exemplary migrations f and g for each listed model difference D such that for any possible variation of m_1, c_1 and c_2, the migrations allow for a successful round-trip migration (with modification). The catalogue is backed by an executable set of test suites, each representing one scenario. Such a suite makes assertions, assuring the above-mentioned properties of f and g when run as a test.

5.3 A Note on Completeness and How to Use This Catalogue

When compiling a catalogue round-trip scenarios, we may consider the concept of *completeness*. In other words, we ask the question whether *this catalogue contains enough exemplary scenarios to compose a round-trip migration strategy for arbitrary model differences?* Simply put, the answer to this question is no. To be more precise, we must further define the term *completeness* in the context of round-trip migrations.

With regard to round-trip migrations, we understand the term *completeness* as follows: Given two data model versions, can the difference between the versions be decomposed into smaller (maybe even atomic) differences, for which fully implemented migration strategies can be found in the catalogue? And further, can the migration strategies of those smaller differences be recomposed into a migration strategy that maintains correctness on a higher level? Authors of related fields (e.g. metamodel co-evolution) argue that such completeness is hard to achieve since higher level changes often entail migration strategies that differ from the simple concatenation of the strategies that fit their atomic constituents [21][25]. A simple example of this, is the model change of *renaming a field* (cf. scenario 1 Rename Field) as opposed to the consecutive *deletion of the old* and *creation of a new* field (cf. scenario 2). Based on this insight, we do not claim the completeness of our scenario catalogue and assume a different standpoint.

Rather than completeness, we aim for a more practical use of the catalogue. It was designed to serve as a knowledge base and guide for the implementation of round-trip migrations. While some scenarios may be applicable exactly as they are presented in this catalogue, others may be more suitable as a demonstration of abstract aspects. As we will see later in our case study, in practice, it often helps to first consider related round-trip scenarios before implementing a concrete migration. Furthermore, we see our catalogue as an initial listing of elementary data model changes, which in the future can be extended by additional and more complex scenarios (e.g. discovery of new scenarios during use, introduction of new modelling features in N4IDL, etc.).

5.4 Catalogue

In this catalogue, we will discuss the following list of scenarios in detail.

No.	Name	Description	trace links	modification detection	Page
1	Rename Field	The name of a field changes.			26
2	Create/Delete Field (functionally independent field)	A new functionally independent field is added/removed to/from a class of the data model.	✓		26
3	Create/Delete Field (functionally 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.	✓		27
4	Create/Delete Reference	A field of reference type is removed from a class of the data model (or added respectively).	✓		29
5	Declare Class as Abstract	A class is declared abstract/concrete.	✓		30
6	Add/Remove a Supertype	A new supertype is declared for a classifier (or removed respectively).	✓		32
7	Generalize/Specialize Field Type	The type of a field is generalized to a supertype or specialized to a subtype respectively.	✓		33
8	Change Field Multiplicity: 0..1 - 1	The multiplicity of a field is specialized from multiplicity `0..1` to `1` or generalized from `1` to `0..1` respectively.	✓	✓	34
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.	✓	✓	35
10	Change Field Multiplicity: 0..n - 1	The multiplicity of a field is specialized from multiplicity `0..n` to `1` or generalized from `1` to `0..n` respectively.	✓	✓	38
11	Pull Up / Push Down Field	A field is pulled up into a superclass (or pushed down respectively).			40
12	Split/Merge Type	Based on a specified criteria, instances of a type of one model version, translate to different (unrelated) types of the other model version.	✓		41
13	Specialize/Generalize Superclass	The supertype of a class is changed to one of the supertype's subclasses/superclasses.	✓		43
14	Extract/Inline Superclass	A new superclass is extracted from the set of fields of an existing type.	✓	✓	45
15	Fold/Unfold Superclass	A new superclass is declared for a type. Common fields of the superclass and the type are then removed from the type (folded into superclass).			46
16	Extract/Inline Subclass	A selection of fields is extracted into a new subclass (or inlined respectively).	✓	✓	47
17	Extract/Inline Class	A selection of fields is extracted into a new delegate class.			49
18	Fold/Unfold Class	A selection of fields is folded into an existing delegate class.			51
19	Collect Field over Reference	A field is collected/pushed over a reference.	✓	✓	52
20	Aggregate Instances	Multiple instances are aggregated into a single primitive value (e.g. computing the average of a value).			54
21	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.			56

Scenario 1: Rename Field

The name of a field changes.

In the example model, the field `field1` is renamed to `field2`.

Data Models

```
export public class RenameField#1 {
        field1 : string
}
```

Version 1

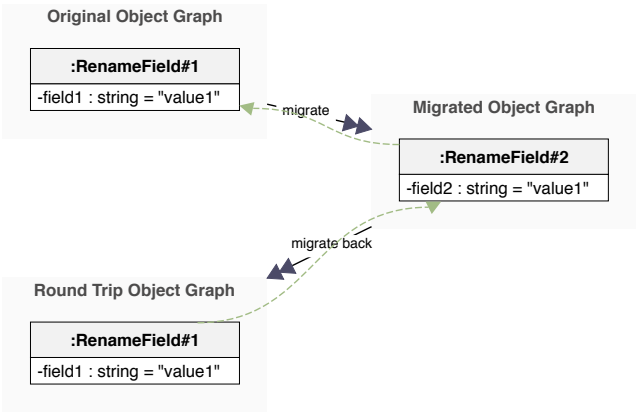
```
export public class RenameField#2 {
        field2 : string
}
```

Version 2

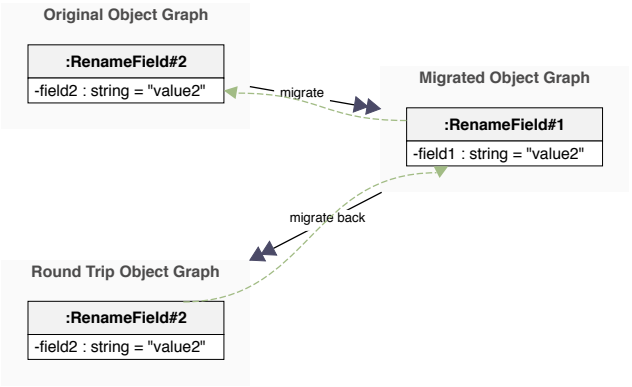
Discussion Since the the data model semantics of both versions are equal except for names, a round-trip migration can be performed without using any traceability features. Furthermore, any modification of `field1` will be reflected by `field2` and vice-versa.

Migrations

```
@Migration
export function migrate(o1 : RenameField#1) : RenameField#2 {
        let o2 = new RenameField#2();

        // renamed field
        o2.field2 = o1.field1;

        return o2;
}

@Migration
export function migrateBack(o2 : RenameField#2) : RenameField#1 {
        let o1 = new RenameField#1();

        // undo field renaming
        o1.field1 = o2.field2;

        return o1;
}
```



Round-Trip 1.1: $\#1 \mapsto \#2 \mapsto \#1$



Round-Trip 1.2: $\#2 \mapsto \#1 \mapsto \#2$

Scenario 2: Create/Delete Field (functionally independent field)

A new functionally independent field is added/removed to/from a class of the data model.

In this scenario we assume, that there do not exist any functional dependencies between `field1` and `field2` in version 2 of the data model.

Data Models

```
export public class AddField#1 {
        public field1 : string
}
```

Version 1

```
export public class AddField#2 {
        public field1 : string
        public field2 : string
}
```

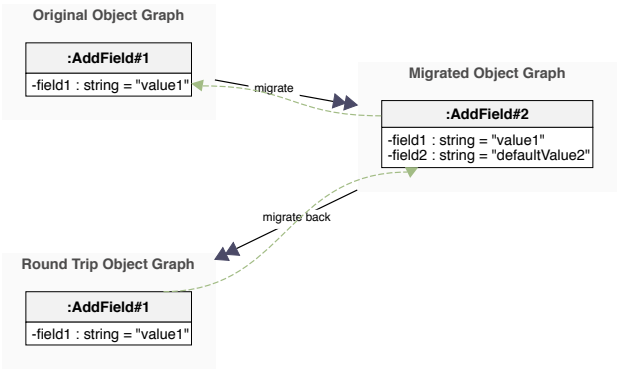
Version 2

Discussion In a $\#1 \mapsto \#2 \mapsto \#1$ RTM, `field2` is set to a default value since the original instance does not provide a value for this field on its own.

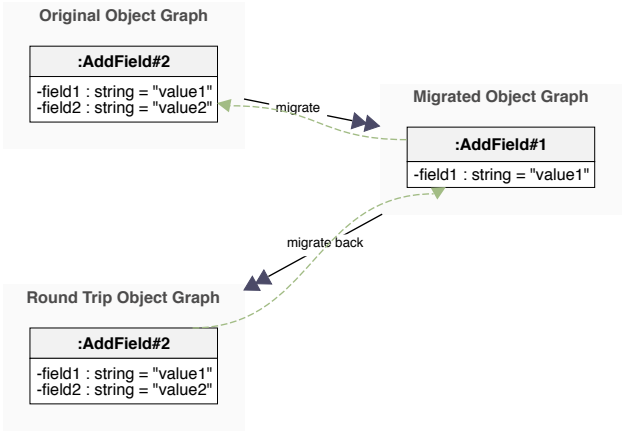
In a $\#2 \mapsto \#1 \mapsto \#2$ RTM, the migration leverages the support for trace links (cf. green links in the RTM graphs), and recovers the value of `field2` from the original instance. A simple copying of the previous value of `field2` is feasible in this scenario, since we assume a functional independence of `field1` and `field2`. Therefore, a potential modification of `field1` in version 2 cannot have any effect on `field2`.

Migrations

```
@Migration
export public function migrate(o1 : AddField#1) : AddField#2 {
        // obtain previous revision
        const previousRevision = context.getTrace(o1)[0] as AddField#2 ||Â {} as AddField#2;

        // instantiate an empty instance
        let o2 = new AddField#2();

        // transfer value for field 'field1'
        o2.field1 = o1.field1;
        // use previous value or a default value alternatively
        o2.field2 = previousRevision.field2 || "defaultValue2";

        return o2;
}

@Migration
export public function migrateBack(o2 : AddField#2) : AddField#1 {
        let o1 = new AddField#1();
        // transfer field value for 'field1'
        o1.field1 = o2.field1;

        return o1;
}
```



Round-Trip 2.1: $\#1 \mapsto \#2 \mapsto \#1$



Round-Trip 2.2: $\#1 \mapsto \#2 \mapsto \#1$

Scenario 3: Create/Delete Field (functionally 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.

In the example data model, version 1 declares a field `hereToStayTwice`. By an informal data invariant, it is specified to always hold a value that equals the concatenation of field `hereToStay` with itself. This implies a functional dependency between the fields. In version 2 of the same class, the field `hereToStayTwice` is removed.

Data Models

```
export public class CreateDeleteDependentField#1 {
        hereToStay : string
        /**
         * The value of this field should always be
         * 'hereToStay', but concatenated with
         * itself twice (e.g. 'A' -> 'A A').
         */
        hereToStayTwice : string
}
```

Version 1

```
export public class CreateDeleteDependentField#2 {
        hereToStay : string
}
```

Version 2

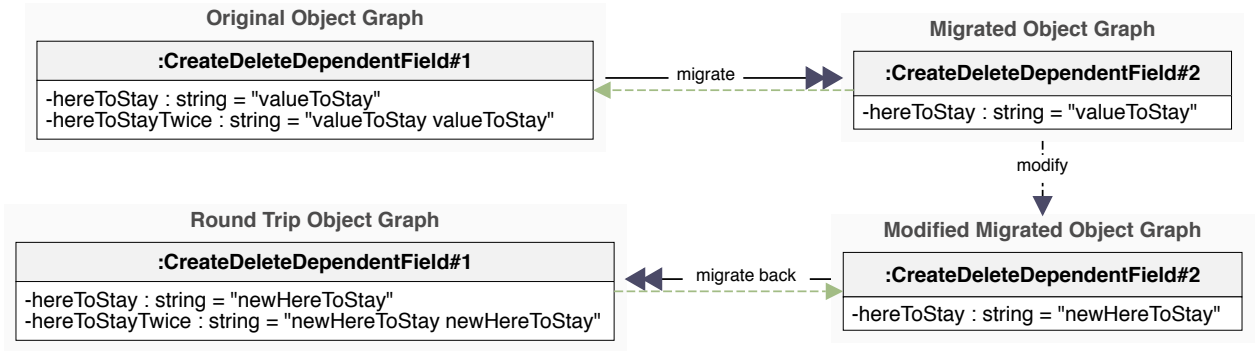
Discussion The migration from version 2 to 1 must consider the potentially new value of field `hereToStay` to compute the corresponding new value of `hereToStayTwice`. Although traceability would allow to access the previous value of `hereToStayTwice`, the migration cannot just copy it, since the functional dependency between `hereToStay` and `hereToStayTwice` may be violated.

For a $\#2 \mapsto \#1 \mapsto \#2$ RTM, the migration computes the missing value of field `hereToStayTwice` from field `hereToStay` in the original instance of version 2. Although `hereToStayTwice` is modified, this modification is not mapped back to the value of `hereToStay` in version 1.

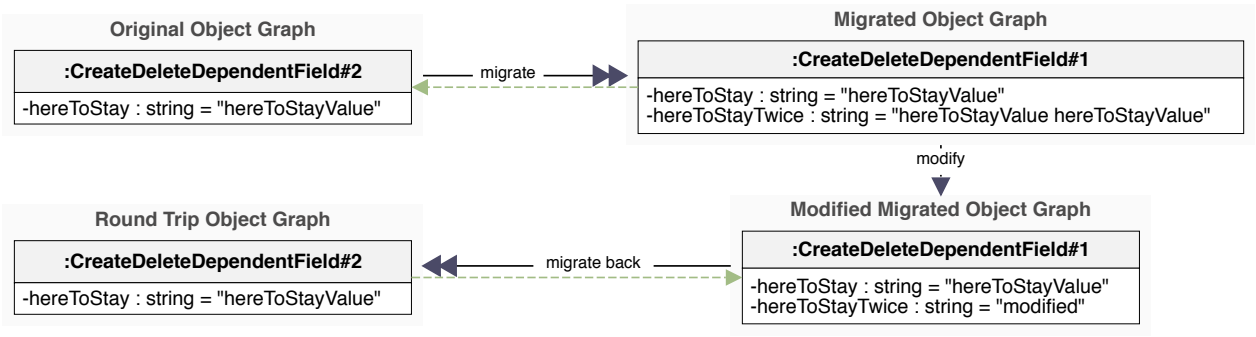
While intuitively this may not seem like a successful round-trip migration, it formally is one according to our initial definition. Nonetheless, the given example demonstrates an issue regarding functional dependencies between fields: A modification of only one of the instances of a redundant bit of information (e.g. the value of field `hereToStayTwice`) causes an inconsistent state, which may in turn result in unexpected behavior. However, this is not an issue with round-trip migrations specifically, but rather with data modeling in general.

Migrations

```
@Migration
export function migrate(o1 : CreateDeleteDependentField#1) : CreateDeleteDependentField#2 {
        let o2 = new CreateDeleteDependentField#2();

        // copy value for field 'hereToStay'
        o2.hereToStay = o1.hereToStay;

        return o2;
}

@Migration
export function migrateBack(o2 : CreateDeleteDependentField#2) : CreateDeleteDependentField#1 {
        let o1 = new CreateDeleteDependentField#1();

        o1.hereToStay = o2.hereToStay
        o1.hereToStayTwice = o2.hereToStay + "␣" + o2.hereToStay

        return o1;
}
```



Round-Trip 3.1: $\#1 \mapsto \#2 \mapsto \#1$ In version 2, the value of field `valueToStay` is modified.



Round-Trip 3.2: $\#2 \mapsto \#1 \mapsto \#2$ In version 1, the value of `hereToStayTwice` is modified.

Scenario 4: Create/Delete Reference

A field of reference type is removed from a class of the data model (or added respectively).

In the example data models, the reference to an instance of type `ReferencedElement` is removed in version 2.

Data Models

```
1  export public class CreateDeleteReference#1 {
2          reference : ReferencedElement
3
4          unrelated : string
5  }
6
7  export public class ReferencedElement#1 {
8          public value : string
9          constructor(@Spec spec : ~i~this) {}
10 }
```

Version 1

```
1  export public class CreateDeleteReference#2 {
2          unrelated : string
3  }
4
5
6  export public class ReferencedElement#2 {
7          public value : string
8          constructor(@Spec spec : ~i~this) {}
9  }
```

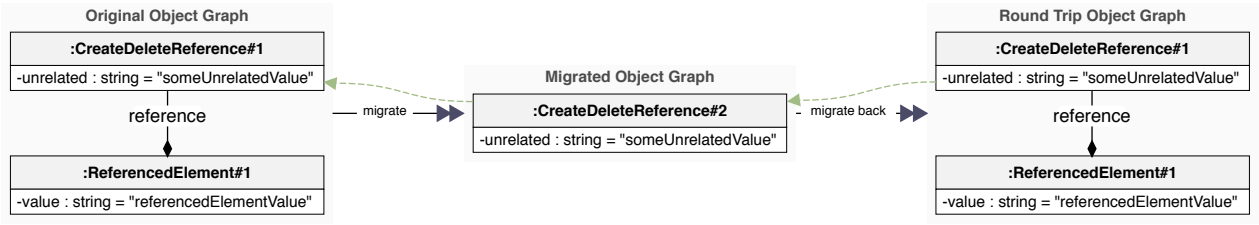
Version 2

Discussion This scenario exhibits similar properties to those of its counterpart for primitively typed fields in scenario 2 Create/Delete Field (functionally independent field) and 3 Create/Delete Field (functionally dependent field). It only differs in that the value which is recovered via trace links, is not of primitive type but an instance of type `ReferencedElement`.

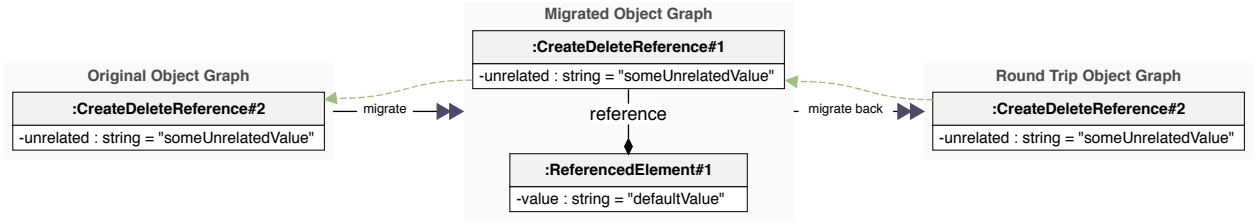
Analogously, this scenario poses the same challenges with regard to potential functional dependencies between different references (cf. discussions in scenario 2 and 3).

Migrations

```
@Migration
export function migrate(o1 : CreateDeleteReference#1) : CreateDeleteReference#2 {
        let o2 = new CreateDeleteReference#2();

        o2.unrelated = o1.unrelated;

        return o2;
}

@Migration
export function migrateBack(o2 : CreateDeleteReference#2) : CreateDeleteReference#1 {
        // obtain previous revision
         const previousRevision = context.getTrace(o2)[0] as CreateDeleteReference#1
                || {} as CreateDeleteReference#1;

        let o1 = new CreateDeleteReference#1();

        o1.unrelated = o2.unrelated;
        // choose a default value for 'reference' or recover previous value
        o1.reference = previousRevision.reference
                || new ReferencedElement#1({value: "defaultValue"});

        return o1;
}
```



Round-Trip 4.1: $\#1 \mapsto \#2 \mapsto \#1$



Round-Trip 4.2: $\#2 \mapsto \#1 \mapsto \#2$

Scenario 5: Declare Class as Abstract

A class is declared abstract/concrete.

In the exemplary data model of this scenario, the class `Value#1` is declared abstract in model version 2. The concrete subclass `SubValue` exists in both versions and is not changed from version 1 to 2.

Data Models

```
export public class MakeClassAbstract#1 {
        public field : Value
}

export public class Value#1 {
        public commonField : string
}

export public class SubValue#1 extends Value {
        public field1 : string
}
```

Version 1

```
export public class MakeClassAbstract#2 {
        public field : Value
}

export public abstract class Value#2 {
        public commonField : string
}

export public class SubValue#2 extends Value {
        public field1 : string
}
```

Version 2

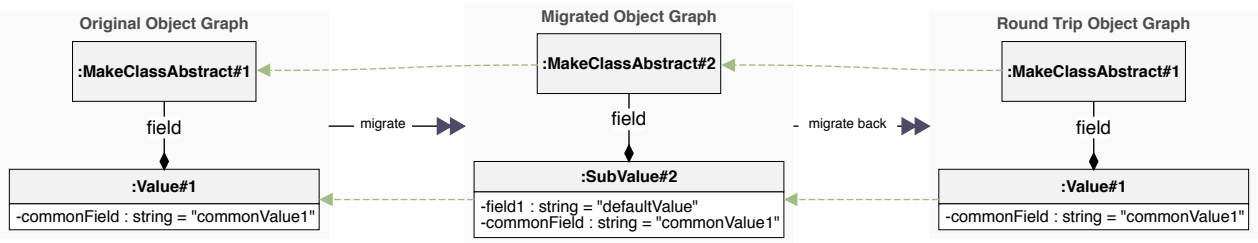
Discussion The core challenge of this scenario lies in the successful migration of instances of `Value#1` to one of its concrete subclasses in version 2. Instances of `SubValue` can be represented in both versions by the identical and correspondingly named types and are thus not of much interest. Concerning the migration of `Value#1` instances however, there are three main points that need to be addressed:

1. A migration must choose a concrete subclass in version 2, which `Value#1` instances are migrated to. Furthermore, default values must be chosen to compensate for missing information. In this example, the migration statically assumes that `Value#1` instances are represented as `SubValue#2` instances in version 2. Assuming there are multiple concrete subclasses to choose from however, it is also possible to make this choice at runtime (e.g. based on a designated field which indicates which type of version 2 to migrate to).
2. In order to successfully round-trip migrate `Value#1` instances, migrations must ensure that `SubValue1#2` instances that originate from a `Value#1` instance, maintain their original type in a round-trip. For instance, a `SubValue#2` instance must always be migrated back to a `Value#1` instance, if it was originally represented as such in version 1 (cf. Round-Trip 5.1). However, this case must be differentiable from `SubValue#2` instances that actually do stem from `SubValue#1` instances. By leveraging the support for traceability, this can be implemented by a runtime type check on the previous revision instance (cf. `migrateBackValue` in the migrations listing).

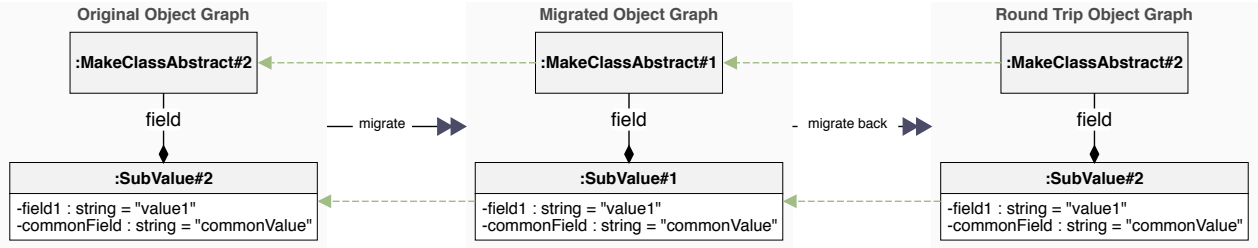
Migrations

```
@Migration
export function migrateValue(o1 : Value#1) : Value#2 {
        const previousRevision = context.getTrace(o1)[0] || {};

        const sv = new SubValue#2();
        // obtain value of field1 from previous revision or choose a default
        sv.field1 = (previousRevision as SubValue#2).field1 || "defaultValue";
        sv.commonField = o1.commonField;
        return sv;
}

@Migration
export function migrateBackValue(o2 : SubValue#2) : Value#1 {
        const previousRevision = context.getTrace(o2)[0] as Value#1
                || {} as Value#1;

        // If in a previous migration we were forced to migrate
        // from Value to SubValue, and 'field1' has not been modified,
        // migrate-back to Value
        if (context.getTrace(o2).length > 0
                && !(previousRevision instanceof SubValue#1)
                && !(context.isModified(o2, "field1"))) {
                const v = new Value#1();
                v.commonField = o2.commonField;
                return v;
        }

        // otherwise just migrate SubValue#2 to an instance of identical type SubValue#1
        const sv = new SubValue#1();
        sv.commonField = o2.commonField;
        sv.field1 = o2.field1;

        return sv;
}
```

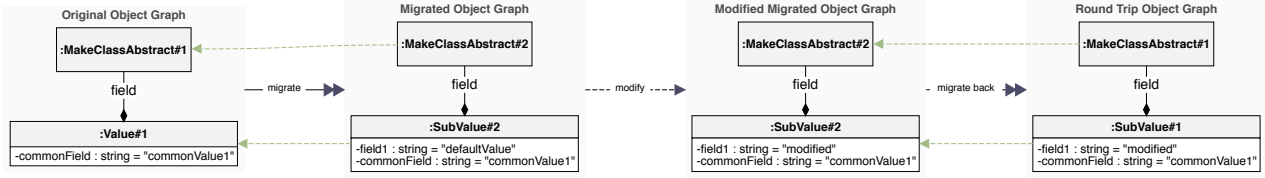
Shortened: For the full implementation of this scenario see the appended source code of the catalogue.



Round-Trip 5.1: $\#1 \mapsto \#2 \mapsto \#1$ Instances of type `Value#1` are migrated to `SubValue#2` by choosing a default value for `field1`.



Round-Trip 5.2: $\#2 \mapsto \#1 \mapsto \#2$ An instance of `SubValue#2` is migrated via `SubValue#1` without any modifications in between.



Round-Trip 5.3: $\#1 \mapsto \#2 \mapsto \#1$ The field `field1` is modified in model version 2. As a consequence its type in the original version is changed to `SubValue#1` in order to represent the modification.

Scenario 6: Add/Remove a Supertype

A new supertype is declared for a classifier (or removed respectively).

In version 2 of the example data model, the class `AddSuperType#1` gains the supertype `SuperType#2`.

Data Models

```
export public class AddSuperType#1 {
        public ownedField : string
}
```

Version 1

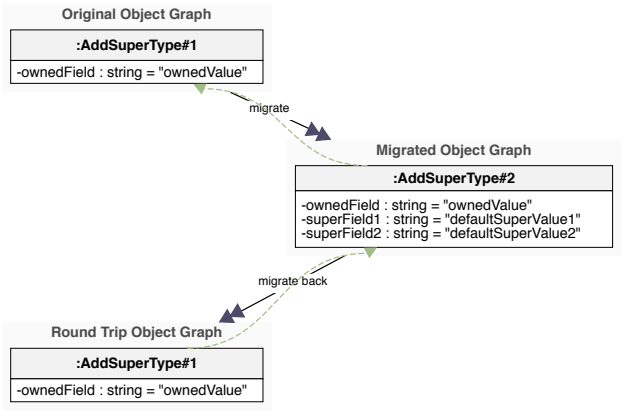
```
export public class AddSuperType#2
        extends SuperType {

        public ownedField : string
}

export public class SuperType#2 {
        public superField1 : string
        public superField2 : string
}
```

Version 2

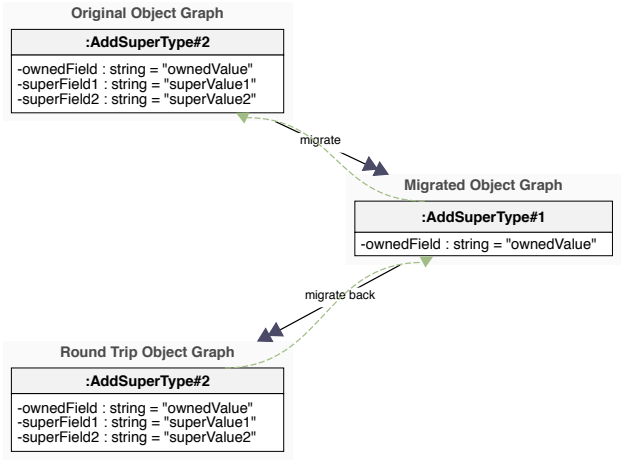
Discussion In version 2 of the data model, instances of type `AddSuperType#2` gain the fields `superField1` and `superField2` which are consumed from the new supertype `SuperType#2`. Therefore, this scenario exhibits the same properties as scenario 2 (Create/Delete Field) for each of the consumed fields. If, in the context of a concrete data model, the consumed fields maintain functional dependencies on existing fields in `AddSuperType#2`, the discussion of scenario 3 may also apply.

Migrations

```
@Migration
export function migrate(o1 : AddSuperType#1) : AddSuperType#2 {
        // obtain previous revision
        const previousRevision = context.getTrace(o1)[0] as AddSuperType#2
                || {} as AddSuperType#2;

        let o2 = new AddSuperType#2();

        o2.ownedField = o1.ownedField;
        o2.superField1 = previousRevision.superField1 || "defaultSuperValue1";
        o2.superField2 = previousRevision.superField2 || "defaultSuperValue2";

        return o2;
}

@Migration
export function migrateBack(o2 : AddSuperType#2) : AddSuperType#1 {
        let o1 = new AddSuperType#1();

        o1.ownedField = o2.ownedField;

        return o1;
}
```



Round-Trip 6.1: $\#1 \mapsto \#2 \mapsto \#1$ The migration chooses default values for the newly introduced fields `superField1` and `superField2` in version 2.



Round-Trip 6.2: $\#1 \mapsto \#2 \mapsto \#1$ Using trace links, the the original values of the new field `superField1` and `superField2` can be restored.

Scenario 7: Generalize/Specialize Field Type

The type of a field is generalized to a supertype or specialized to a subtype respectively.

In the example model, the type of field `field` is generalized from `FieldType` to the supertype `SuperFieldType`. The types `FieldType` and `SuperFieldType` remain unchanged from version 1 to version 2.

Data Models

```
export public class GeneralizeFieldType#1 {
        public field : FieldType
}

export public class SuperFieldType#1 {
        public generic : string
}

export public class FieldType#1
        extends SuperFieldType {

        public specific : string
}
```

Version 1

```
export public class GeneralizeFieldType#2 {
        public field : SuperFieldType
}

export public class SuperFieldType#2 {
        public generic : string
}

export public class FieldType#2
        extends SuperFieldType {

        public specific : string
}
```

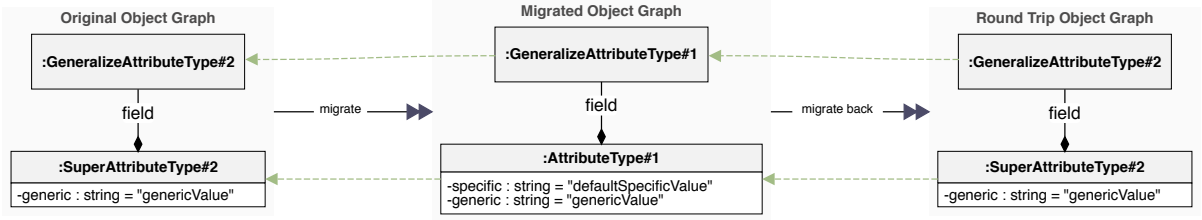
Version 2

Discussion This scenario exhibits similar properties to those of scenario 5 Declare Class as Abstract. This is due to the fact that this scenario requires the migration of an instance of more specific type to an instance of less specific type (cf. `FieldType` vs. `SuperFieldType`). The migrations need to make use of traceability information to differentiate between the different cases of `FieldType` instances in version 1 (cf. Fig. 7.1 vs Fig. 7.2). These are those that originally stem from a `FieldType` and those migrated from the supertype. Furthermore, a modification of field `specific` may further affect the migration strategy (cf. Fig. 7.3). See scenario 5 for a thorough discussion.

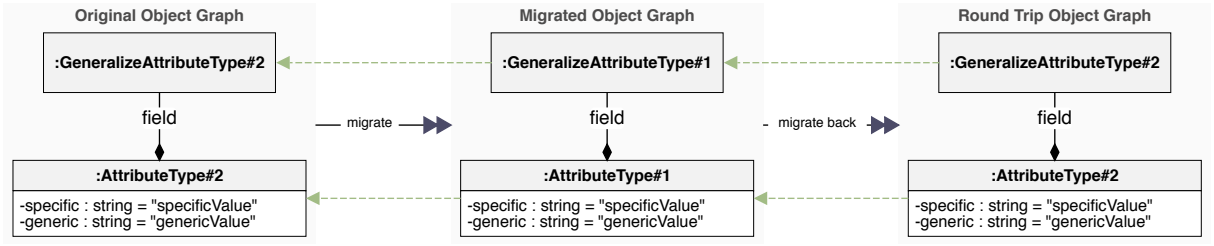
Migrations

```
@Migration
export function migrateBackSuperAttribute(a2 : SuperFieldType#2) : FieldType#1 {
        // obtain previous revision
        const previousRevision = context.getTrace(a2)[0] as FieldType#1 || {} as FieldType#1;

        const a1 = new FieldType#1();
        a1.generic = a2.generic;
        a1.specific = previousRevision.specific ||Â "defaultSpecificValue";

        return a1;
}

@Migration
export function migrateAttribute(a1 : FieldType#1) : SuperFieldType#2 {
        const previousRevision = (context.getTrace(a1)[0] || {}) as SuperFieldType#2;

        let a2 : SuperFieldType#2

        // If field has been of type SuperAttributeType but not
        // of AttributeType, and field 'specific' has not been changed
        if (previousRevision instanceof SuperFieldType#2
                && !(previousRevision instanceof FieldType#2)
                && !(context.isModified(a1, "specific"))) {
                // migrate back to SuperAttribute type
                a2 = new SuperFieldType#2();
        } else { // no previous revision, or a previous revision of type AttributeType
                a2 = new FieldType#2();
                (a2 as FieldType#2).specific = a1.specific;
        }

        // set generic field in any case
        a2.generic = a1.generic;

        return a2;
}
```

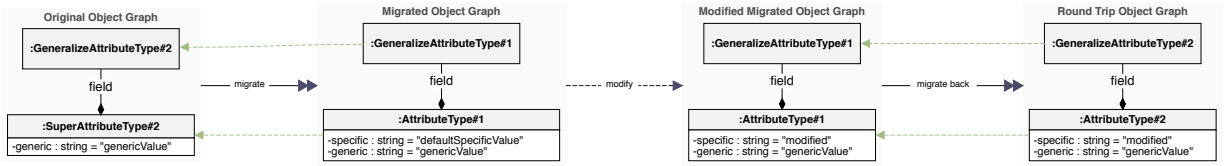
Shortened: For the full implementation of this scenario see the appended source code of the catalogue.



Round-Trip 7.1: $\#2 \mapsto \#1 \mapsto \#2$ with initial field type `SuperFieldType`.



Round-Trip 7.2: $\#2 \mapsto \#1 \mapsto \#2$ with initial field type `FieldType`.



Round-Trip 7.3: $\#2 \mapsto \#1 \mapsto \#2$ with a modification of field `specific`.

Scenario 8: Change Field Multiplicity: 0..1 - 1

The multiplicity of a field is specialized from multiplicity `0..1` to `1` or generalized from `1` to `0..1` respectively.

In the example data model, the mandator field `field` of model version 1 is declared optional in version 2.

Data Models

```
export public class GeneralizeAttributeOptional#1 {
        public field : string
}
```

Version 1

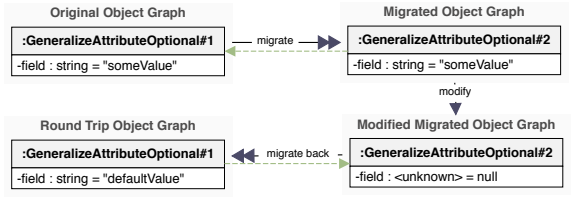
```
export public class GeneralizeAttributeOptional#2 {
        public field? : string
}
```

Version 2

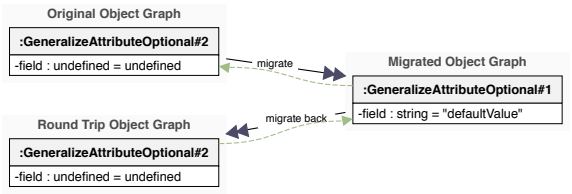
Discussion Model version 2 is more generic, as it allows for additional instances that hold a `null`-value for field `field`. To accommodate for this difference, a migration must choose a default value for `field` when migrating a `null`-value from model version 2 to 1. In a $\#2 \mapsto \#1 \mapsto \#2$ round-trip, it must further be assured, that a default value is not translated back into version 2 but rather detected and therefore mapped to the original `null`-value (cf. Round-Trip 8.1).

Migrations

```
@Migration
export function migrate(o1 : GeneralizeAttributeOptional#1) : GeneralizeAttributeOptional#2 {
        let o2 = new GeneralizeAttributeOptional#2();

        const previousRevision = context.getTrace(o1)[0] as GeneralizeAttributeOptional#2

        // If 'field' has not been modified, and a previous revision could
        // be obtained
        if (previousRevision != undefined
                && !context.isModified(o1, "field")) {
                // re-use the previous value for 'field'
                o2.field = previousRevision.field;
        } else {
                // otherwise copy over the value in 'field'
                o2.field = o1.field;
        }

        return o2;
}

@Migration
export function migrateBack(o2 : GeneralizeAttributeOptional#2) : GeneralizeAttributeOptional#1 {
        let o1 = new GeneralizeAttributeOptional#1();

        // use default value in case 'o2.field' is null
        o1.field = o2.field || "defaultValue";

        return o1;
}
```



Round-Trip 8.1: $\#1 \mapsto \#2 \mapsto \#1$ A modification in version 2 sets `field` to `null`. This results in a default value in version 1.



Round-Trip 8.2: $\#2 \mapsto \#1 \mapsto \#2$ In version 1, `field` is set to a default value, which is later not translated back to model version 2.

Scenario 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.

In the exemplary data model, the type of field `field` is changed from an optional reference to an instance of type `Element` to an array with element type `Element`.

Data Models

```
1 export public class GeneralizeAttributeOptional2Array#1 {
2         public field? : Element
3 }
4 export public class Element#1 {
5         public value : string
6 }
```

Version 1

```
1 export public class GeneralizeAttributeOptional2Array#2 {
2         public field : Array<Element>
3 }
4 export public class Element#2 {
5         public value : string
6 }
```

Version 2

Discussion The general migration strategy we propose for this scenario, is to introduce a mapping of the value of `field` to a specific index in the array in model version 2 (See 20 Aggregate Instances for an alternative). For the sake of simplicity, we will assume for further discussion that this designated array index is always the first element (0). However, it may of course be any other index or even a dynamically chosen index.

For this scenario, there are two main observations to be made:

1. The migrations need to translate any change to the designated index in the array of model version 2 to version 1 and vice-versa. Since an array may be mutated using common operators such as insert and delete, we must consider the situation in which the array does not hold an element for that specific index. Therefore, this scenario can be seen as an instance of scenario 8 where we migrate between a mandatory and an optional field. This migration from version 2 to 1 can be implemented using the following mappings:

`field` in version 2	**`field` in version 1**
Empty Array	`field` is set to null.
Array without an element at the designated index.	`field` is set to null.
Array with an instance of `Element#2` at the designated index.	`field` is set to the migrated equivalent of the element at the designated index.

   Note that with this strategy we bind the value of `field` in version 1 to an index in the array of version 2, not a specific element. For instance, when a new element is inserted at index 0 of the array in version 2, this new element will replace the current value of `field` in version 1 (cf. Round-Trip 9.1). Another instance of this behavior is demonstrated in Round-Trip 9.4.

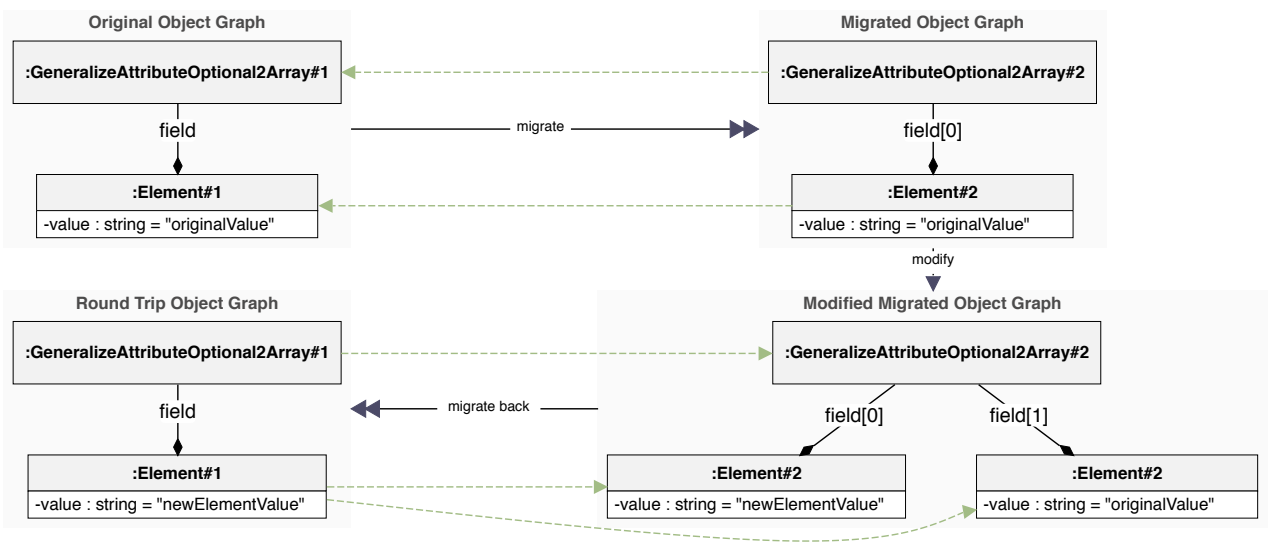
2. When round-trip migrating an instance of version 2 via version 1, a migration needs to update the designated array element with the potentially changed value of `field` from version 1. Apart from that, it needs to *restore all other array elements using trace links* (see line 17-19 in `migrate`).

The Round-Trips 9.2, 9.3 and 9.4 demonstrate how this migration strategy maps common modifications (set null, delete element, insert element) of `field` in version 1 and 2 to the other model version.

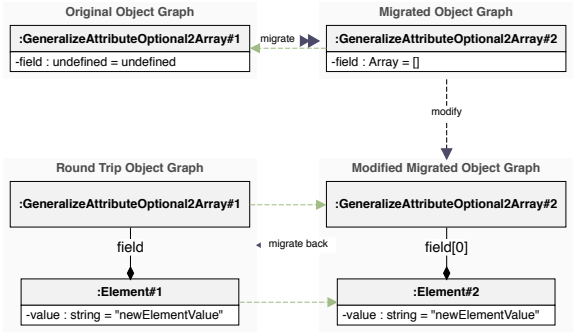
Migrations

```
@Migration
export function migrate(o1 : GeneralizeAttributeOptional2Array#1) : GeneralizeAttributeOptional2Array#2 {
        // obtain previous revision
        const previousRevision = context.getTrace(o1)[0] as GeneralizeAttributeOptional2Array#2
                || {} as GeneralizeAttributeOptional2Array#2;

        let o2 = new GeneralizeAttributeOptional2Array#2();

        let elements : Array<Element#2> = [];

        // if o1.field is present
        if (o1.field != null) {
                // add migrated field value as first array element
                elements.push(Migrations.copy(Element#2, o1.field));
        }

        // add all previousRevision elements, but the first one
        (previousRevision.field || []).slice(1)
                .forEach(e => elements.push(Migrations.copy(Element#2, e)));

        o2.field = elements;

        return o2;
}

@Migration
export function migrateBack(o2 : GeneralizeAttributeOptional2Array#2) : GeneralizeAttributeOptional2Array#1 {
        let o1 = new GeneralizeAttributeOptional2Array#1();

        // migrate the element at index 0, if not present set 'o1.field' to null
        o1.field = Migrations.migrateElementAt(o2.field, 0, null);

        return o1;
}
```

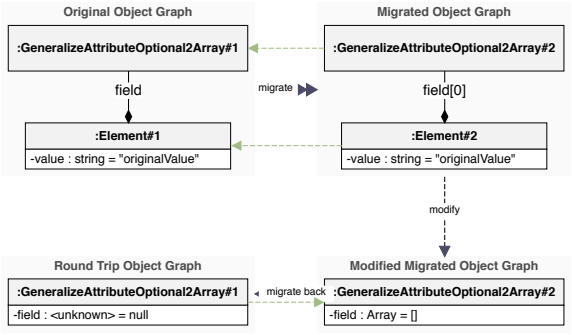
Shortened: For the full implementation of this scenario see the appended source code of the catalogue.



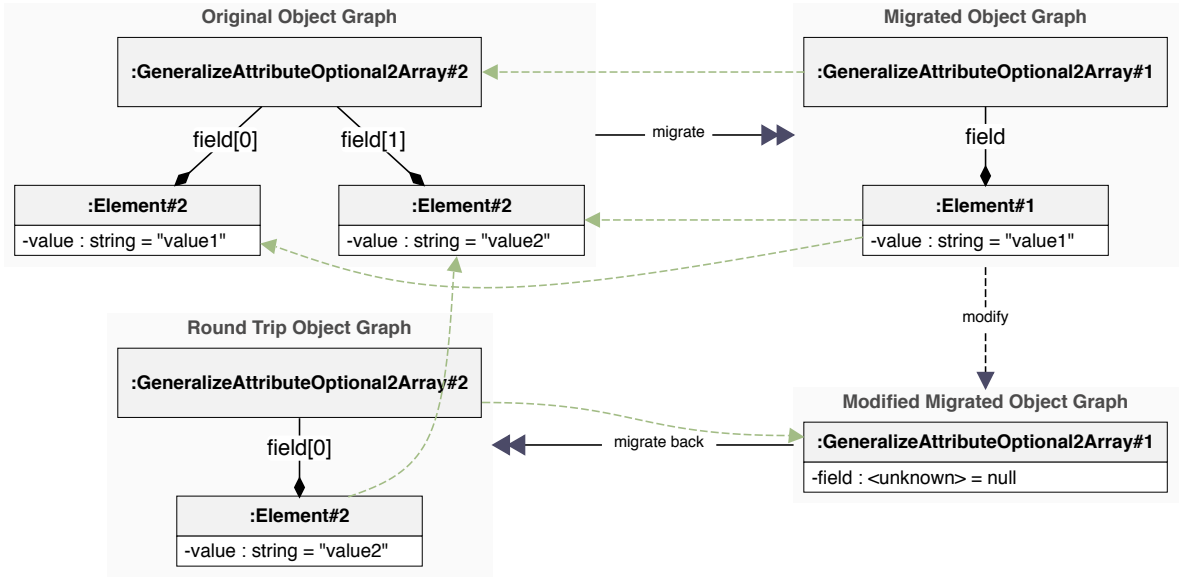
Round-Trip 9.1: $\#1 \mapsto \#2 \mapsto \#1$ In version 2, a new element is inserted into the array at index 0. As a consequence, the new element of the array replaces the value of `field` in version 1.



Round-Trip 9.2: $\#2 \mapsto \#1 \mapsto \#2$ An absent value of `field` in version 1 translates to an empty array in version 2. Adding a new element to the array in version 2, results in setting the value of `field` in version 1.



Round-Trip 9.3: $\#2 \mapsto \#1 \mapsto \#2$ Removing all elements from the array of version 2, translates to setting `field` in version 1 to null.



Round-Trip 9.4: $\#2 \mapsto \#1 \mapsto \#2$ Setting `field` to null in version 1, results in removing the corresponding element from the array of version 2.

Scenario 10: Change Field Multiplicity: 0..n - 1

The multiplicity of a field is specialized from multiplicity `0..n` to `1` or generalized from `1` to `0..n` respectively.

In our example data model, the type of field `field` is changed from `Element` to `Array<Element>`.

Data Models

```
1 export public class GeneralizeAttributeArray#1 {
2         public field : Element
3 }
4 export public class Element#1 {
5         public value : string
6 }
```

Version 1

```
1 export public class GeneralizeAttributeArray#2 {
2         public field : Array<Element>
3 }
4 export public class Element#2 {
5         public value : string
6 }
```

Version 2

Discussion This scenario exhibits similar characteristics to those of scenario 9. Therefore, we propose an analogous migration strategy of introducing a mapping of the value of `field` in version 1 to a designated index in the corresponding array in model version 2.

As opposed to scenario 9 however, model version 1 does not allow to set `field` to null when the corresponding array index does not hold a value at migration time. Instead, we propose the use of default values. The following mapping between states of `field` in version 1 and 2 may then be used:

`field` in version 2	**`field` in version 1**
Empty Array	Default Value for `Element#1`
Array without an element at the designated index.	Default Value for `Element#1`
Array with an instance of `Element#2` at the designated index.	Migrated version of the instance.

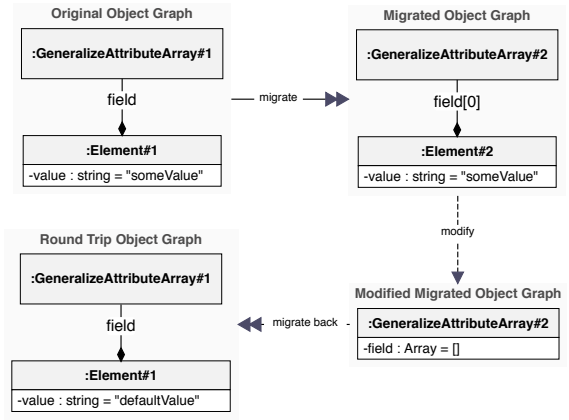
Similar to other scenarios in which we resort to default values, we need to make sure that a default instance for `field` is not translated back into version 2. Instead, we want to assure that we restore the original value of `field` using trace links. See line 11 to 14 in `migrate` for an exemplary implementation. Round-Trip 10.2 demonstrates such a situation.

Based on this migration strategy, we inherit a similar behavior as in scenario 9 (cf. Round-Trip 10.1 and 10.3). Overall however, we rate this scenario to be slightly more complex. Since in version 1 the field is mandatory, we rely on default values which in turn require the use of modification detection. This significantly increases the complexity of the migration code.

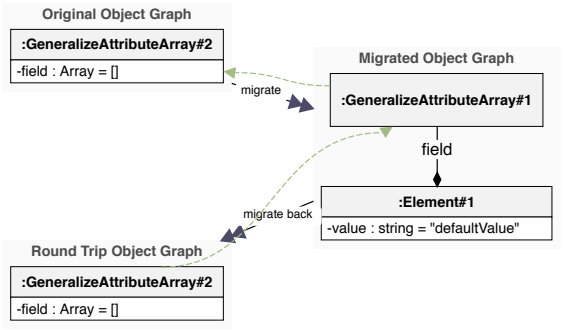
Migrations

```
1  @Migration
2  export function migrate(o1 : GeneralizeAttributeArray#1)
3          : GeneralizeAttributeArray#2 {
4          let o2 = new GeneralizeAttributeArray#2();
5
6          let elements : Array<Element#2> = [];
7
8          const previousRevision = context.getTrace(o1)[0] as GeneralizeAttributeArray#2;
9
10         // detect default value of 'o1.field'
11         if (previousRevision !== undefined
12                 && previousRevision.field.length == 0 && !context.isModified(o1, "field")) {
13                 // restore empty array, if default value in 'o1.field' remained unmodified
14                 elements = [];
15         } else {
16                 // add migrated field value as first array element
17                 elements.push(migrate(o1.field));
18
19                 // if a previous revision can be obtained
20                 if (previousRevision != undefined) {
21                         // add all previousRevision elements, but the first one
22                         previousRevision.field.slice(1)
23                                 .forEach(e => elements.push(Migrations.copy(Element#2, e)));
24                 }
25         }
26
27         // Finally assign the array of migrated elements to
28         // the migrated instance field 'field'
29         o2.field = elements;
30
31         return o2;
32 }
33 @Migration
34 export function migrateBack(o2 : GeneralizeAttributeArray#2) : GeneralizeAttributeArray#1 {
35         let o1 = new GeneralizeAttributeArray#1();
36
37         // migrate only the first element of the array
38         o1.field = Migrations.migrateElementAt(o2.field, 0, createDefaultElement());
39
40         return o1;
41 }
```

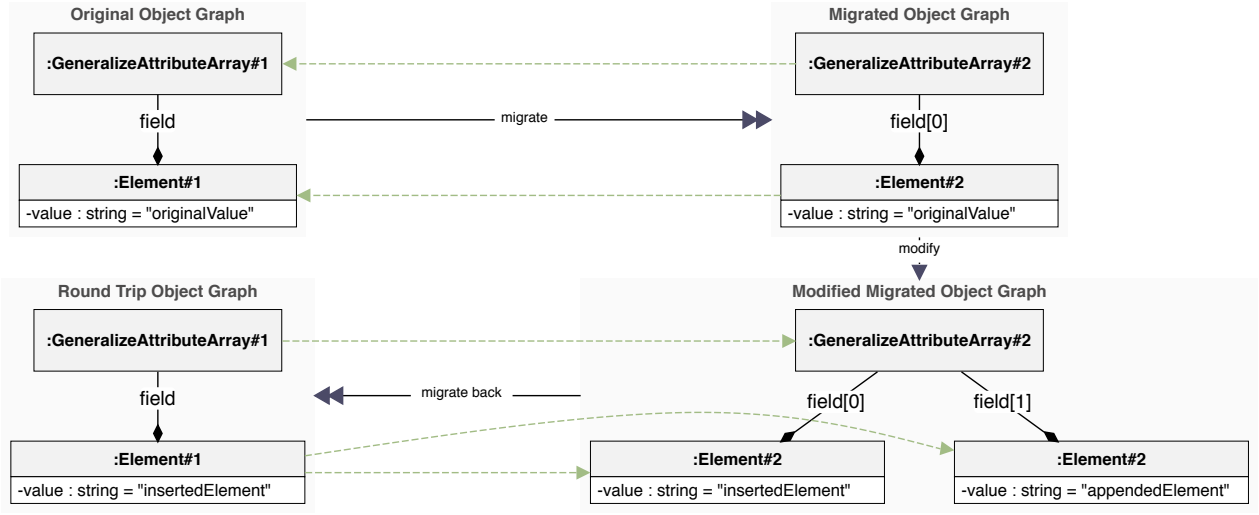
Shortened: For the full implementation of this scenario see the appended source code of the catalogue.



Round-Trip 10.1: $\#1 \mapsto \#2 \mapsto \#1$ In version 2 the only element in array *field* is removed. This results in a default instance of `Element#1` in version 1.



Round-Trip 10.2: $\#2 \mapsto \#1 \mapsto \#2$ An empty array is migrated to a default instance of `Element#1`. When migrating back, the (unchanged) default value is recognized and the original empty array is restored.



Round-Trip 10.3: $\#1 \mapsto \#2 \mapsto \#1$ In version 2, a new element is inserted into the array at index 0. As a consequence, the new element of the array replaces the value of `field` in version 1.

Scenario 11: Pull Up / Push Down Field

A field is pulled up into a superclass (or pushed down respectively).

In our exemplary data model, the field `f` is pulled up into the supertype `SuperClass` in version 2 of the model.

Data Models

```
1 export public class SuperClass#1 {}
2 export public class PullUpFeature#1 extends SuperClass {
3         public f : string
4 }
```

Version 1

```
1 export public class SuperClass#2 {
2         public f : string
3 }
4 export public class PullUpFeature#2 extends SuperClass {}
```

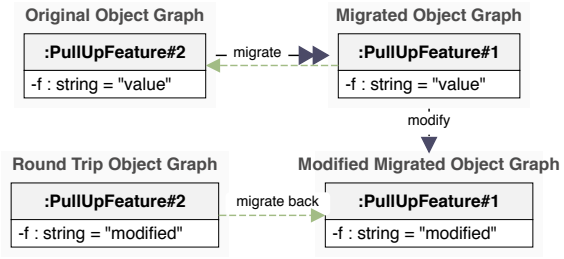
Version 2

Discussion On an instance level, the origin of field `f` (super field or local field) is not of importance since all fields of the supertype are consumed. Therefore we may migrate instances of `PullUpFeature` by copying.

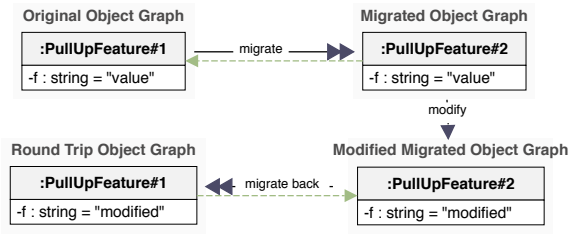
Another change that can be observed in this scenario, is that in model version 2, the class `SuperClass` gains the field `f`. This must be handled separately in the corresponding migration for type `SuperClass` (see scenarios 2, 3, 4 for a discussion).

Migrations

```
@Migration
export public function migratePullUpFeature(o1 : PullUpFeature#1) : PullUpFeature#2 {
        const o2 = new PullUpFeature#2();

        // simple copy the value of 'f'
        o2.f = o1.f;

        return o2;
}

@Migration
export public function migrateBackPullUpFeature(o2 : PullUpFeature#2) : PullUpFeature#1 {
        const o1 = new PullUpFeature#1();

        // simple copy the value of 'f'
        o1.f = o2.f;

        return o1;
}
```



Round-Trip 11.1: $\#2 \mapsto \#1 \mapsto \#2$ Instance of type `PullUpFeature` can simply be migrated by copying. Modifications equally apply in both model version.



Round-Trip 11.2: $\#1 \mapsto \#2 \mapsto \#1$ A round-trip in the other direction almost looks identical except for the type versions.

Scenario 12: Split/Merge Type

Based on a specified criteria, instances of a type of one model version, translate to different (unrelated) types of the other model version.

In the example data model, the type `Combined` indicates by a `type` field, which of the optional fields `intValue` and `stringValue` hold the actual data [2]. In other words, it models the concept of union types. In model version 2, `Combined#2` is split into two separate types. The referring class `SplitClass#2` furthermore specializes the type of its field `f`, as it now only allows instances of type `IntValue#2`.

Data Models

```
export public class SplitClass#1 {
        public f : Combined
}
export public enum CombinedType#1 { INT, STRING }
export public class Combined#1 {
        public type : CombinedType
        public stringValue? : string
        public intValue? : int
}
```

Version 1

```
export public class SplitClass#2 {
        public f : IntValue
}

export public class IntValue#2 {
        public intValue : int
}

export public class StringValue#2 {
        public stringValue : string
}
```

Version 2

[2] At this point we assume that the informal contract of the `type` field of `Combined#1` is not violated (e.g. `stringValue` is null even though `type` is `STRING`.

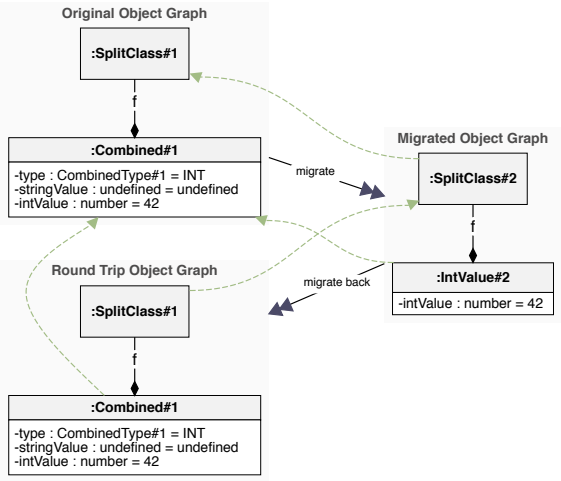
Discussion Since migrations operate on the instance level, we may evaluate the value of the `type` field during migration. Based on this information, we can decide whether to use the value in `intValue` of `Combined#1` (line 32) or whether we must provide a default value (line 29/30). Migrating instances of type `IntValue#2` back to `Combined#1` can be implemented by setting the `type` field to the corresponding `INT` literal (e.g. line 40).

Since we are dealing with default values, we must detect those and replace them with the previous revision if no modification can be detected (cf. line 16).

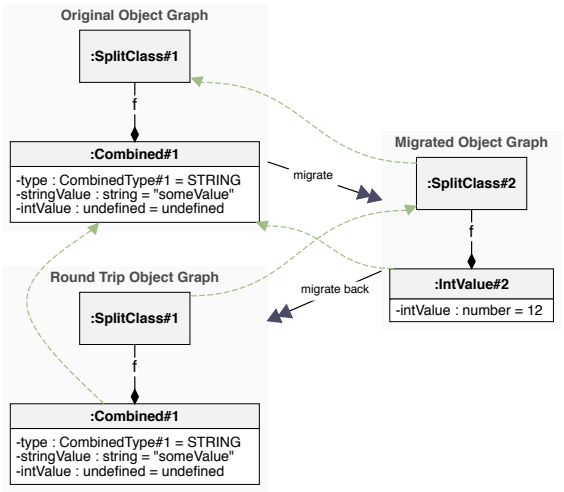
Migrations

```
1  @Migration
2  function migrate(sc1 : SplitClass#1) : SplitClass#2 {
3          const sc2 = new SplitClass#2();
4
5          // delegate migration of 'f'
6          sc2.f = migrate(sc1.f);
7
8          return sc2;
9  }
10
11 @Migration
12 function migrateBack(sc2 : SplitClass#2) : SplitClass#1 {
13         const sc1 = new SplitClass#1();
14         const previousRevision = context.getTrace(sc2)[0] as SplitClass#1;
15
16         if (previousRevision && !context.isModified(sc2.f)) {
17                 sc1.f = copy(Combined#1, previousRevision.f);
18         } else {
19                 // migrate instance of IntValue to Combined by delegation
20                 sc1.f = migrate(sc2.f);
21         }
22         return sc1;
23 }
24
25 @Migration
26 function migrateIntValue(combined : Combined#1) : IntValue#2 {
27         const iv = new IntValue#2();
28         if (combined.type != CombinedType#1.INT) {
29                 // choose default value if 'combined' is of wrong type
30                 iv.intValue = 12;
31         } else {
32                 iv.intValue = combined.intValue;
33         }
34         return iv;
35 }
36
37 @Migration
38 function migrateBackIntValue(iv : IntValue#2) : Combined#1 {
39         const c = new Combined#1();
40         c.type = CombinedType#1.INT;
41         c.intValue = iv.intValue;
42         return c;
43 }
```

Shortened: For the full implementation of this scenario see the appended source code of the catalogue.



Round-Trip 12.1: $\#1 \mapsto \#2 \mapsto \#1$ In case the instance of `Combined#1` represents an integer value, the migration can be performed without the use of any default values.



Round-Trip 12.2: $\#1 \mapsto \#2 \mapsto \#1$ If the instance of `Combined#1` does not represent an integer value, the migration strategy makes use of default values in model version 2.

Scenario 13: Specialize/Generalize Superclass

The supertype of a class is changed to one of the supertype's subclasses/superclasses.

In the example data model, the supertype of class `SpecializeSuperType` is specialized to `SuperType` from `SuperSuperType` in version 2 of the model.

Data Models

```
1  export public class SuperSuperType#1 {
2          // no fields
3  }
4
5  export public class SuperType#1 extends SuperSuperType {
6          public superField : string
7  }
8
9  export public class SpecializeSuperType#1
10         extends SuperSuperType {
11         // Inheriting all fields of SuperSuperType,
12         // therefore this type does not have
13         // any fields in version 1.
14 }
```

Version 1

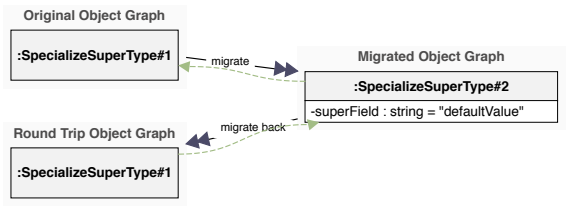
```
1  export public class SuperSuperType#2 {
2          // no fields
3  }
4
5  export public class SuperType#2 extends SuperSuperType {
6          public superField : string
7  }
8
9  export public class SpecializeSuperType#2
10         extends SuperType {
11         // Inheriting all fields of HighestSuperType,
12         // therefore this type inherits field 'superField'
13 }
```

Version 2

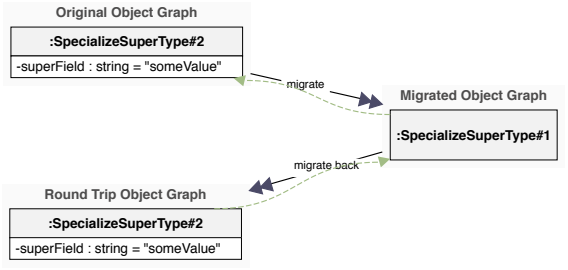
Discussion Since the specialization/generalization of the supertype can be seen as the removal and the addition of two unrelated supertypes, the migration strategy of this scenario is similar to that of scenario 6 Add/Remove a Supertype. Therefore, the actually migrated change is the creation/deletion of fields. Depending on potential functional dependencies between existing fields and fields that are introduced by the changed supertype, the scenarios 2, 3, 4 are then applicable.

Migrations

```
@Migration
function migrate(sst1 : SpecializeSuperType#1) : SpecializeSuperType#2 {
        const sst2 = new SpecializeSuperType#2();
        const previousRevision = context.getTrace(sst1)[0] as SpecializeSuperType#2;

        if (previousRevision) {
                // re-use previous revision 'superField' value, if present
                sst2.superField = previousRevision.superField;
        } else {
                // otherwise, choose a default value for 'superField'
                sst2.superField = "defaultValue";
        }

        return sst2;
}

@Migration
function migrateBack(sst2 : SpecializeSuperType#2) : SpecializeSuperType#1 {
        // empty type, nothing to migrate
        return new SpecializeSuperType#1();
}
```



Round-Trip 13.1: $\#1 \mapsto \#2 \mapsto \#1$ For the newly introduced field `superField`, a default value is chosen.



Round-Trip 13.2: $\#2 \mapsto \#1 \mapsto \#2$ If a previous revision can be obtained via trace links, the original value of `superField` is restored.

Scenario 14: Extract/Inline Superclass

A new superclass is extracted from the set of fields of an existing type.

In the example data model, in version 2 the field `genericField` is extracted into the new superclass `SuperClass#2`.

Data Models

```
export public class ExtractSuperClass#1 {
        public specificField : string
        public genericField : string
}
```

Version 1

```
export public class SuperClass#2 {
        public genericField : string
}

export public class ExtractSuperClass#2 extends SuperClass {
        public specificField : string
}
```

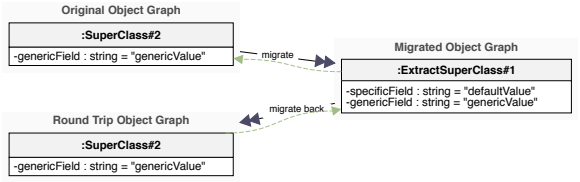
Version 2

Discussion Similar to scenario 11 Pull Up / Push Down Field, the migration for this scenario can be performed by copying, since the data model change is not visible on an instance level. However, by extracting a class, a new class is added to the data model. Therefore, we must also consider the migration of `SuperType#2` instances back to model version 1 (cf. Round-Trip 14.1 and 14.3). With the exemplary migration strategy below, we propose a similar solution as in scenario 5 *Declare class as abstract*.

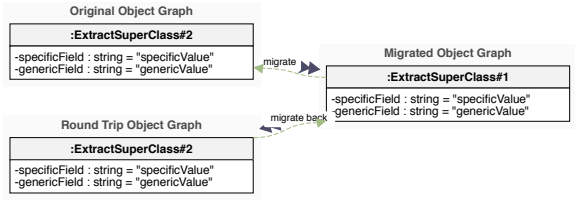
Alternatively, we can avoid the migration of `SuperType#2` instances, by additionally *declaring the extracted superclass as abstract*. As a consequence, the data model change of this scenario becomes fully transparent from an instance perspective (semantically equivalent data models as defined in Def. 2.8).

Migrations

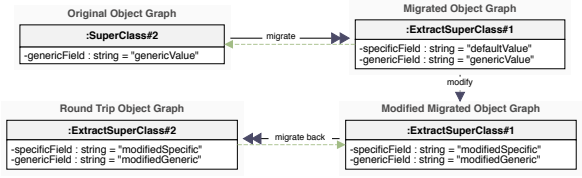
```
@Migration
function migrate(e1 : ExtractSuperClass#1) : SuperClass#2 {
        const previousRevision = context.getTrace(e1)[0] as SuperClass#2;

        // if a previous revision can be obtained
        if (previousRevision &&
                // and e1 has originally been an instance of SuperClass ...
                (previousRevision instanceof SuperClass#2) &&
                !(previousRevision instanceof ExtractSuperClass#2) &&
                // and e1.specificField has not been modified (is default)
                !(context.isModified(e1, "specificField"))) {

                // ... migrate back to a SuperClass instance
                const s = new SuperClass#2();
                s.genericField = e1.genericField;
                return s;
        }

        // otherwise copy all values over to a new instance
        // of type ExtractSuperClass#2
        return copy(ExtractSuperClass#2, e1);
}

@Migration
function migrateBack(e2 : ExtractSuperClass#2) : ExtractSuperClass#1 {
        // copy all values over to new instance of type ExtractSuperClass#1
        return copy(ExtractSuperClass#1, e2);
}

@Migration
function migrateSuperClass(s : SuperClass#2) : ExtractSuperClass#1 {
        // migrate instances of SuperClass#2 back to ExtractSuperClass#1
        // as there is no other means to represent them in model version 1
        const e = new ExtractSuperClass#1();

        e.genericField = s.genericField;
        e.specificField = "defaultValue";

        return e;
}
```



Round-Trip 14.1: $\#2 \mapsto \#1 \mapsto \#2$ An instance of `SuperType#2` is migrated to an instance of `ExtractSuperClass` in version 1. For the additional field `specificField`, a default value must be chosen.



Round-Trip 14.2: $\#1 \mapsto \#2 \mapsto \#1$ On an instance-level, moving fields into a superclass is not visible.



Round-Trip 14.3: $\#2 \mapsto \#1 \mapsto \#2$ A modification of `specificField` of an `ExtractSuperClass#1` instance that originally stems from a `SuperType#2` instance, is mapped back to an instance of `ExtractSuperClass` in version 2.

Scenario 15: Fold/Unfold Superclass

A new superclass is declared for a type. Common fields of the superclass and the type are then removed from the type (folded into superclass).

In our exemplary data model, the class `FoldSuperType` gains the new supertype `SuperClass`. As a consequence its fields `f1` and `f2` can be folded into `SuperClass`.

Data Models

```
1 export public class SuperClass#1 {
2         public f1 : string
3         public f2 : string
4 }
5 export public class FoldSuperClass#1 {
6         public f1 : string
7         public f2 : string
8 }
```

Version 1

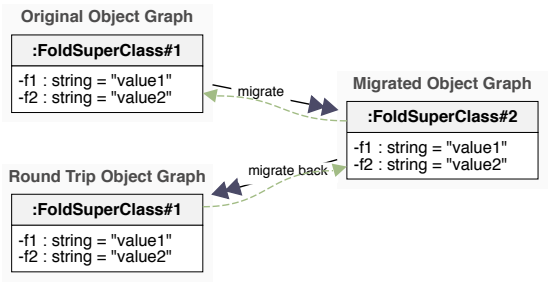
```
1 export public class SuperClass#2 {
2         public f1 : string
3         public f2 : string
4 }
5 export public class FoldSuperClass#2 extends SuperClass {}
```

Version 2

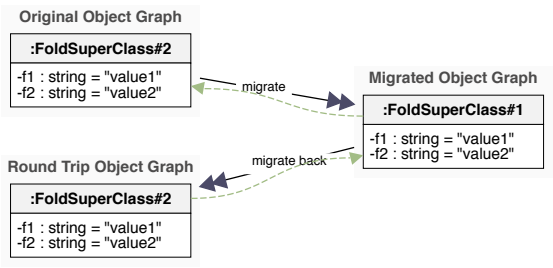
Discussion This scenario is closely related to scenario 14 Extract Super Class. It only differs in the fact that the superclass already existed in model version 1. Therefore, there is no need to further consider a migration of `SuperClass#2` instances. Since the changes to the class hierarchy are transparent on the instance level, a migration-by-copying strategy can be deployed.

Migrations

```
1  @Migration
2  function migrateFoldSuperClass(f : FoldSuperClass#1) : FoldSuperClass#2 {
3          const fs2 = new FoldSuperClass#2();
4
5          // simply copy over values of 'f1' and 'f2'
6          fs2.f1 = f.f1;
7          fs2.f2 = f.f2;
8
9          return fs2;
10 }
11
12 @Migration
13 function migrateBackFoldSuperClass(f : FoldSuperClass#2) : FoldSuperClass#1 {
14         const fs1 = new FoldSuperClass#1();
15
16         // simply copy over values of 'f1' and 'f2'
17         fs1.f1 = f.f1;
18         fs1.f2 = f.f2;
19
20         return fs1;
21 }
```



Round-Trip 15.1: $\#1 \mapsto \#2 \mapsto \#1$ On an instance-level, the changes in the class hierarchy are transparent.



Round-Trip 15.2: $\#2 \mapsto \#1 \mapsto \#2$ Apart from the concrete type versions, this round-trip equals the other direction.

Scenario 16: Extract/Inline Subclass

A selection of fields is extracted into a new subclass (or inlined respectively).

In our example data model, the field `specificField` is extracted into a new subclass `SubA#2` of type `A`. The field `genericField` remains part of the original type `A`. The class `ExtractSubClass` only serves as a container that holds a reference to an instance of type `A`.

Data Models

```
1  export public class A#1 {
2          public specificField : string
3          public genericField : string
4  }
5
6  export public class ExtractSubClass#1 {
7          public f1 : A
8  }
```

Version 1

```
1  export public class A#2 {
2          public genericField : string
3  }
4
5  export public class SubA#2 extends A {
6          public specificField : string
7  }
8
9  export public class ExtractSubClass#2 {
10         public f1 : A
11 }
```

Version 2

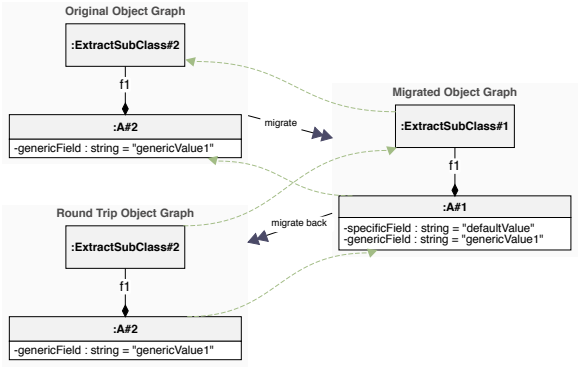
Discussion Since references to instances of type `A#2` can always refer to an instance of type `SubA#2` as well, migrating from model version 1 to 2, can simply be implemented by migrating all instances of `A#1` to instances of `SubA#2`.

For $\#2 \mapsto \#1 \mapsto \#2$ round-trips however, we must consider the case of direct instances of `A#2` (cf. Round-Trip 16.1 and 16.2). Similar to scenario 5 Declare Class as Abstract, we need to make use of traceability features in order to detect `A#1` instances with default values that have previously been `A#2` instances (line 7-14 in `migrateA`).

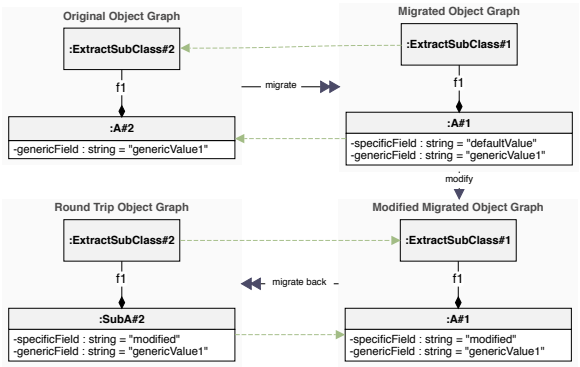
Migrations

```
@Migration
function migrateA(a : A#1) : A#2 {
        const previousRevision = context.getTrace(a)[0] as A#2;

        let a2 : A#2;

        // if previous revision exists, is of type SubA...
        if (previousRevision instanceof SubA#2 ||
                // ..or 'specificField' has been modified
                context.isModified(a, "specificField")) {

                // migrate-back to an instance of SubA
                a2 = new SubA#2();
                (a2 as SubA#2).specificField = a.specificField;
        } else {
                // otherwise migrate-back to A
                a2 = new A#2();
        }

        // copy over value of 'genericField'
        a2.genericField = a.genericField;

        return a2;
}

@Migration
function migrateBackA(a : A#2) : A#1 {
        const a1 = new A#1();

        a1.genericField = a.genericField;
        // use default value for missing field 'specificField'
        a1.specificField = "defaultValue";

        return a1;
}

@Migration
function migrateBackSubA(a : SubA#2) : A#1 {
        const a1 = new A#1();

        a1.genericField = a.genericField;
        a1.specificField = a.specificField;

        return a1;
}
```

Shortened: For the full implementation of this scenario see the appended source code of the catalogue.



Round-Trip 16.1: $\#2 \mapsto \#1 \mapsto \#2$ An instance of A#2 is migrated to A#1 using a default value. When migrating back however, the default value is detected and the instance is again represented as A#2



Round-Trip 16.2: $\#2 \mapsto \#1 \mapsto \#2$ An instance of A#2 is migrated to A#1 using a default value. After a modification of field `specificField`, the instance is migrated back to SubA#2 to represent the modification in version 2.

Scenario 17: Extract/Inline Class

A selection of fields is extracted into a new delegate class.

In the example data model, the field `f` is moved from class `ExtractClass` to class `DelegateClass`. Instead, `ExtractClass` holds a mandatory reference to an instance of `DelegateClass` in version 2 of the data model.

Data Models

```
1 export public class ExtractClass#1 {
2         public f : string
3 }
```

Version 1

```
1 export public class ExtractClass#2 {
2         public delegate : DelegateClass
3 }
4 export public class DelegateClass#2 {
5         public f : string
6 }
```

Version 2

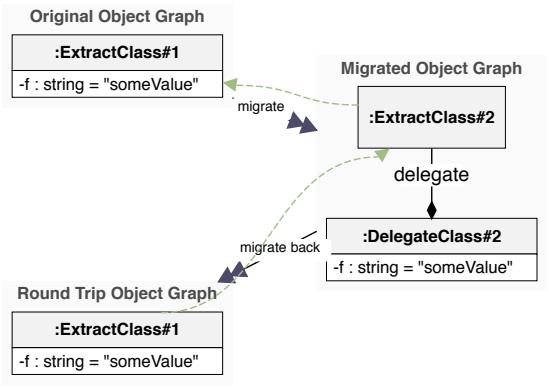
Discussion Since the field `delegate` is mandatory, a migration can collect the value of field `f` via the `delegate` reference (line 17 in `migrateBackExtractClass`). This means, that version 2 of the data model is semantically equivalent to model version 1 and no traceability features are required to successfully migrate instances.

In model version 2, instances of the new class `DelegateClass` may occur. In our exemplary migration implementation, we do not implement the migration of such `DelegateClass` instances back to model version 1, since we assume that those only occur in combination with `ExtractClass` instances. In a concrete case, this assumption may be invalid when there exists another use of `DelegateClass` instances. Depending on the nature of such, further scenarios must then be consulted.

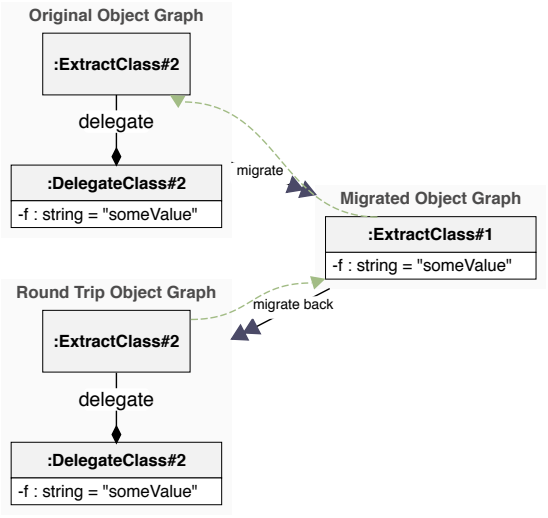
A modification of field `f` can simply be mapped in both directions, by applying it to the corresponding field of `DelegateClass` or `ExtractClass` respectively.

Migrations

```
1  @Migration
2  function migrateExtractClass(ec : ExtractClass#1) : ExtractClass#2 {
3          const ec2 = new ExtractClass#2();
4
5          // create new delegate class to hold value of 'field''
6          ec2.delegate = new DelegateClass#2();
7          ec2.delegate.f = ec.f;
8
9          return ec2;
10 }
11
12 @Migration
13 function migrateBackExtractClass(ec : ExtractClass#2) : ExtractClass#1 {
14         const ec1 = new ExtractClass#1();
15
16         // collect value of 'field' via 'delegate''
17         ec1.f = ec.delegate.f;
18
19         return ec1;
20 }
```



Round-Trip 17.1: $\#1 \mapsto \#2 \mapsto \#1$ The value of field `f` is moved into a new `DelegateClass` instance.



Round-Trip 17.2: $\#2 \mapsto \#1 \mapsto \#2$ The value of field `f` in model version 1 is collected from the `DelegateClass` instance of model version 2.

Scenario 18: Fold/Unfold Class

A selection of fields is folded into an existing delegate class.

In the example data model, the field `f` is folded into class `OtherClass`. Instead, `FoldClass` holds a reference to delegate class `OtherClass` in version 2 of the data model.

Data Models

```
export public class OtherClass#1 {
        public f : string
}

export public class FoldClass#1 {
        public f : string
}
```

Version 1

```
export public class OtherClass#2 {
        public f : string
}

export public class FoldClass#2 {
        public delegate : OtherClass
}
```

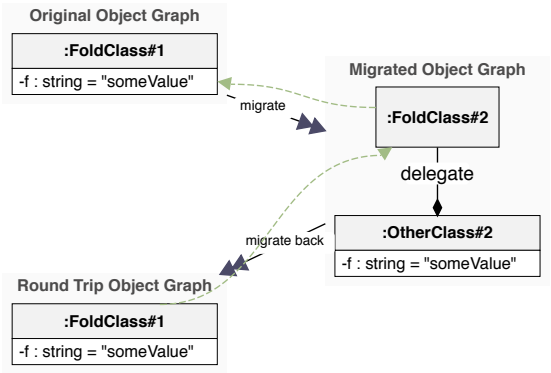
Version 2

Discussion Similar to scenario 17 Extract/Inline Class, the two data model versions in the scenario represent semantically equivalent models. Thus, a round-trip migration can be performed without the use of any traceability features (see scenario 17 for further discussion).

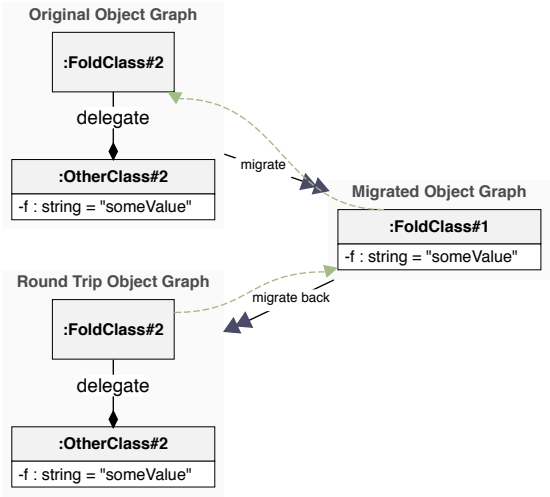
Since the class into which the fields are folded, already existed in version 1 of the model, we do not have to further consider the migration of `OtherClass` instances.

Migrations

```
@Migration
function migrateFoldClass(fc : FoldClass#1) : FoldClass#2 {
        const fc2 = new FoldClass#2();

        fc2.delegate = new OtherClass#2();
        // move value of 'f' to delegate instance
        fc2.delegate.f = fc.f;

        return fc2;
}
@Migration
function migrateBackFoldClass(fc : FoldClass#2) : FoldClass#1 {
        const fc1 = new FoldClass#1();

        // collect value of 'f' from delegate instance
        fc1.f = fc.delegate.f;

        return fc1;
}
```



Round-Trip 18.1: $\#1 \mapsto \#2 \mapsto \#1$ The value of field `f` is moved to delegate class `OtherClass#2`.



Round-Trip 18.2: $\#2 \mapsto \#1 \mapsto \#2$ The value of field `f` is collected from delegate class `OtherClass#2`.

Scenario 19: Collect Field over Reference

A field is collected/pushed over a reference.

In our example data model, `field` is collected from `SuperClass` via field `reference` into the class `CollectField`. As a consequence, `field` of `CollectField#2` assumes the multiplicity (0..1) of field `reference`.

Data Models

```
export public class CollectField#1 {
        public reference? : SourceClass
}
export public class SourceClass#1 {
        public field : string
        public someOtherField : string
}
```

Version 1

```
export public class CollectField#2 {
        public reference? : SourceClass
        public field? : string
}

export public class SourceClass#2 {
        public someOtherField : string
}
```

Version 2

Discussion In a migration strategy for this scenario, we mainly need to decide on a mapping between potential `null`-values for the optional fields `CollectField#1.reference`, CollectField#2.reference and `CollectField#2.field`. Specifically, for a migration of `CollectField` from model version 2 to 1 we need to consider a trade-off between default values and loss of information. In our exemplary migration strategy, we propose the following mapping:

Version 2		**Version 1**
`CollectField#2.reference`	`CollectField#2.field`	
`null`	value	Set `reference` to an instance of `SourceClass` and choose a default value for `SourceClass#1.someOtherField`.
value	`null`	Migrate the instance of `SourceClass#2` of field `reference` to version 1 by choosing a default value for `SourceClass#1.field`.
value	value	Migrate the instance of `SourceClass#2` of field `reference` to version 1 using the value in `field`.
`null`	`null`	Set `CollectField#1.reference` to `null`.

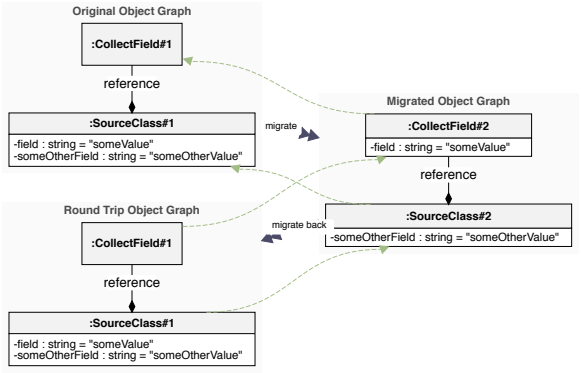
Round-Trip 19.1, 19.2, 19.3 and 19.4 demonstrate how this strategy affects different cases. In general, we aim to minimize the use of default values while also minimizing the loss of information (e.g. `field` is set to a value in one version, but the change is not visible in the other version).

The overall challenge this scenario poses, is the combinations of multiplicities in `SourceClass` as well as of field `reference`. Since both fields of `SourceClass` are mandatory, they are coupled. More specifically, we cannot only represent the presence of one of the fields. In model version 2 however, the fields are decoupled in that we may represent the presence of only one of `field` and `someOtherField`. In our migration strategy, we propose to compensate for this semantic difference, by the use of default values.

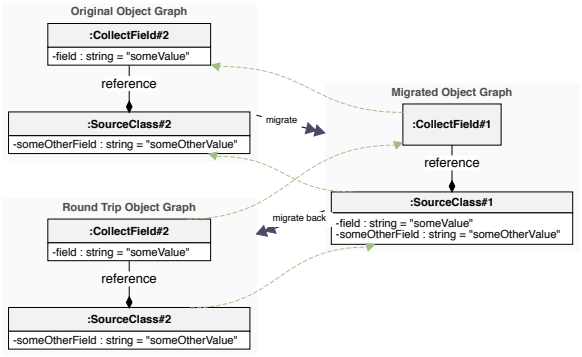
Migrations

```
@Migration function migrateCollectField(cf : CollectField#1) : CollectField#2 {
	const cf2 = new CollectField#2();

	// migrate value of 'field', if 'reference' is present
	cf2.field = cf.reference == null ? null : cf.reference.field;

	if (cf.reference == null) {
		cf2.reference = null;
	} else {
		const previousRevision = context.getTrace(cf)[0] as CollectField#2;
		// If in the previous revision 'reference' was null
		// and 'reference.someOtherField' holds the unmodified default value...
		if (previousRevision &&
			previousRevision.reference == null &&
			!context.isModified(cf.reference, "someOtherField")) {
			// ... migrate back to 'reference' being null
			cf2.reference = null;
		} else {
			// otherwise, 'someOtherField' holds new information (changed))
			cf2.reference = migrate(cf.reference);
		}
	}

	return cf2;
}

@Migration
function migrateBackSourceClass(s : SourceClass#2, fieldValue : string) : SourceClass#1 {

	const sc1 = new SourceClass#1();

	// migrate 'someOtherField'
	sc1.someOtherField = s.someOtherField;
	// use given value for 'field' or a default value
	sc1.field = fieldValue || "defaultValue";

	return sc1;
}

@Migration function migrateBackCollectField(cf : CollectField#2) : CollectField#1 {
	const cf1 = new CollectField#1();

	if (cf.reference != null) {
		cf1.reference = migrate(cf.reference, cf.field);
	} else {
		if (cf.field != null) {
			cf1.reference = new SourceClass#1();
			cf1.reference.field = cf.field;
			cf1.reference.someOtherField = "someOtherDefaultValue";
		} else {
			cf1.reference = null;
		}
	}

	return cf1;
}
```

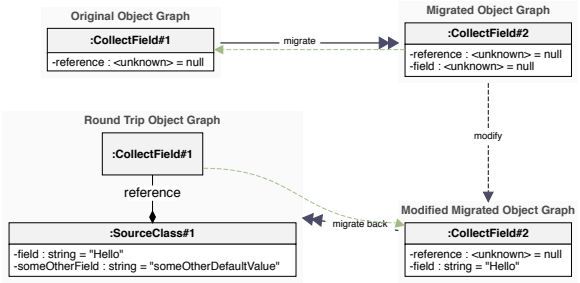
Shortened: For the full implementation of this scenario see the appended source code of the catalogue.



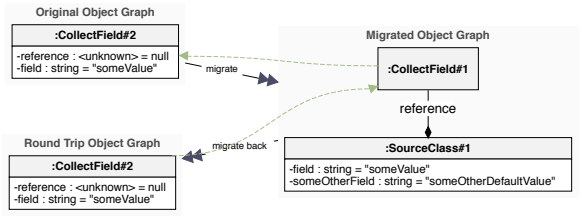
Round-Trip 19.1: $\#1 \mapsto \#2 \mapsto \#1$ The value of `field` is collected into class `CollectField` in model version 2.



Round-Trip 19.2: $\#2 \mapsto \#1 \mapsto \#2$ Via `reference`, `field` is pushed back to class `SourceClass` in model version 1.



Round-Trip 19.3: $\#1 \mapsto \#2 \mapsto \#1$ A `null`-value for `reference` in model-version 1 translates to `null`-values for both `field` and `reference` in model version 2. A modification sets a value for `field` only. As a consequence, a default value is chosen for `someOtherField` in version 1 of the model.



Round-Trip 19.4: $\#2 \mapsto \#1 \mapsto \#2$ The use of a `SourceClass` default instance is detected when migrating back to model version 2.

Scenario 20: Aggregate Instances

Multiple instances are aggregated into a single primitive value (e.g. computing the average of a value).

In our example data model, a list of `Exam` instances, which each have a field for an individual `mark`, are aggregated into an average mark value in model version 2 (cf. `Course#2.averageMark`).

Data Models

```
1 export public class Exam#1 {
2         public mark : number
3 }
4
5 export public class Course#1 {
6         public exams : Array<Exam>
7 }
```

Version 1

```
1 export public class Course#2 {
2         public averageMark : number
3 }
```

Version 2

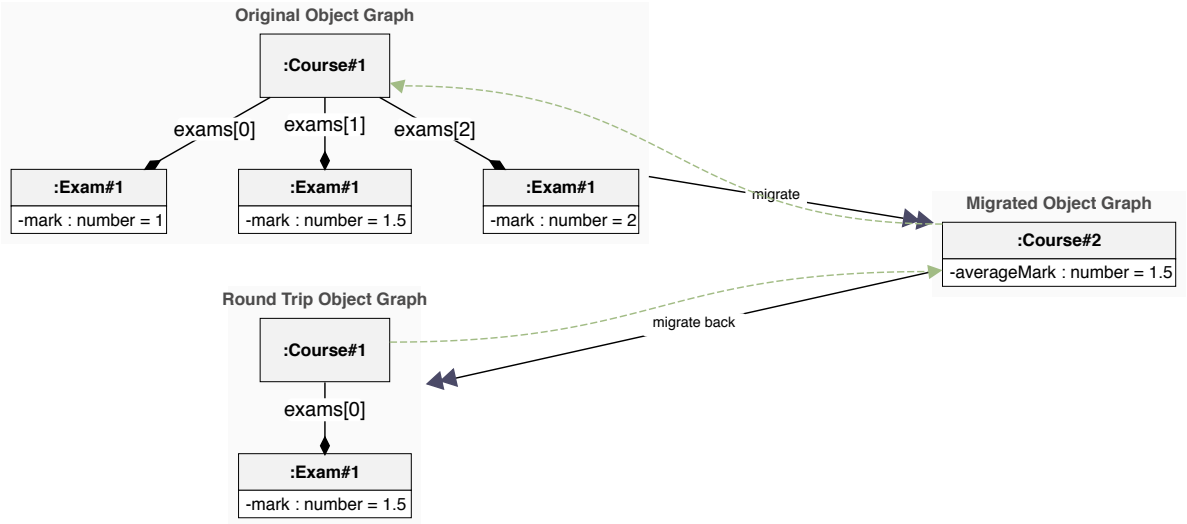
Discussion A migration from version 1 to 2 can easily be implemented by computing the average mark in the migration. However, a migration in the other direction cannot always be performed successfully. While we could leverage traceability information to restore the original list of exams, a modification of field `averageMark` cannot be translated back into model version 1. The reason for this, is the non-injective character of the average operation. Given an average value, it is not possible to reconstruct the different components that formed the operands of the operation. Therefore, it is not clear how to translate a modified average mark in version 2 back to the list of exams in version 1.

Below, we propose a migration strategy that allows for limited correctness according to our definition of successful round-trip migrations (cf. Def. 2.7). In particular, a $\#2 \mapsto \#1 \mapsto \#2$ round-trip (cf. Round-Trip 20.2) is successful. Nonetheless, a loss of information cannot be avoided, as the round-trips 20.1 and 20.3 illustrate.

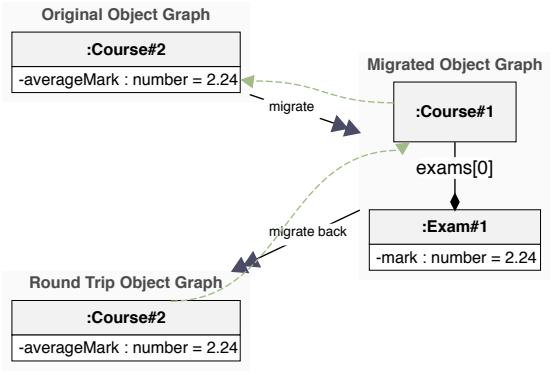
Another possible strategy to resolve the core problem of this scenario, is to declare the `averageMark` field in data model version 2 as `@Final` (read-only). This change on the model level circumvents the problem of mapping back modifications of field `averageMark` to the original model version. Furthermore, a read-only constraint renders the complete migration from version 2 to version 1 obsolete. Since in version 2 no modification can be performed, we may then directly re-use the original instance when migrating back.

Migrations

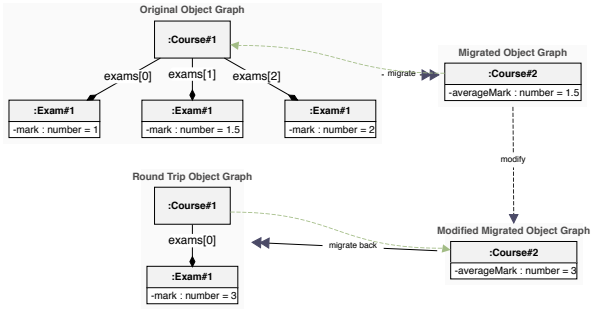
```
@Migration function migrateCourse(c : Course#1) : Course#2 {
        const c2 = new Course#2();

        // compute average mark of all exams
        c2.averageMark = (c.exams.map(e => e.mark) as [number])
                .reduce((acc, m) => acc + m, 0) / c.exams.length;

        return c2;
}

@Migration function migrateBackCourse(c : Course#2) : Course#1 {
        const c1 = new Course#1();

        c1.exams = [ new Exam#1()Â ]
        c1.exams[0].mark = c.averageMark;

        return c1;
}
```



Round-Trip 20.1: $\#1 \mapsto \#2 \mapsto \#1$ With multiple exams in version 1 of the model, the round-trip migration is not successful as information on the individual exam marks is lost.



Round-Trip 20.2: $\#2 \mapsto \#1 \mapsto \#2$ With our proposed migration strategy, this round-trip can be performed successfully.



Round-Trip 20.3: $\#1 \mapsto \#2 \mapsto \#1$ A modification of the `averageMark` in version 2, leads to the creation of a single exam with that exact mark. All information on individual marks is lost.

Scenario 21: 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.

In the example data model, in version 2 the type X is split into Y and Z. Its field a is moved to Y and its field b is moved to Z. The type `SplitField#1` holds a reference to an X instance in version 1 and to corresponding instances of Y and Z in version 2.

Data Models

```
export public class SplitFields#1 {
        public x : X
}

export public class X#1 {
        public a : string
        public b : string
}
```

Version 1

```
export public class SplitFields#2 {
        public y : Y
        public z : Z
}
export public class Y#2 {
        public a: string
}

export public class Z#2 {
        public b: string
}
```

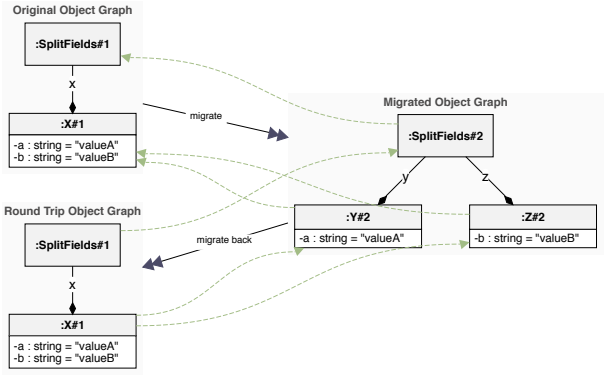
Version 2

Discussion The one-to-one correspondence of an instance of X and instances of Y and Z implies a semantic equivalence between the data model versions. Therefore, we may deploy a migrations strategy which does not use any traceability features. Furthermore, our proposed migration strategy leverages the support for multiple migration parameters and return types in N4IDL. This also becomes apparent in the Round-Trip 21.1 and 21.2.

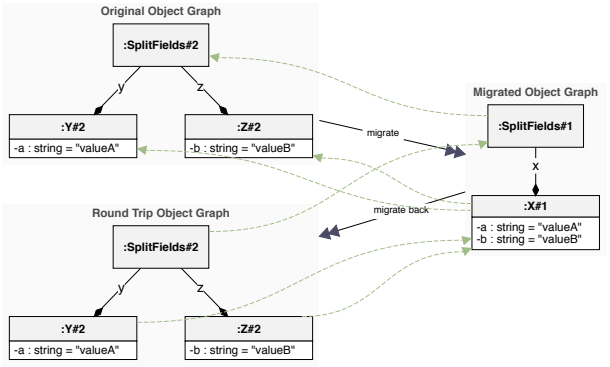
Due to the one-to-one correspondence of the fields in type X and the fields in Y and Z, any modifications of the fields a and b, can directly be mapped to the other version.

Migrations

```
@Migration function migrateSplitFields(sf : SplitFields#1) : SplitFields#2 {
        const sf2 = new SplitFields#2();

        // migrating X#1 by splitting it into Y#2, Z#2
        const yAndZ = migrate(sf.x);
        sf2.y = yAndZ.y;
        sf2.z = yAndZ.z;

        return sf2;
}

@Migration function migrateBackSplitFields(sf : SplitFields#2) : SplitFields#1 {
        const sf1 = new SplitFields#1();

        // migrate back to X#1 based on Y#2, Z#2
        sf1.x = migrate(sf.y, sf.z);

        return sf1;
}

@Migration function migrateYZ(y : Y#2, z : Z#2) : X#1 {
        const x = new X#1();

        x.a = y.a;
        x.b = z.b;

        return x;
}

@Migration function migrateX(x : X#1) : ~Object with {y : Y#2, z : Z#2} {
        const y = new Y#2();
        const z = new Z#2();

        // copy over the values of field 'a' and 'b' to
        // the instances of X and Z respectively
        y.a = x.a;
        z.b = x.b;

        return {y: y, z: z};
}
```



Round-Trip 21.1: $\#1 \mapsto \#2 \mapsto \#1$ The fields a and b are distributed between the instances of Y and Z in model version 2.



Round-Trip 21.2: $\#1 \mapsto \#2 \mapsto \#1$ Due to the use of migrations with multiple parameters and return types, the instances of Y and Z both link back to the same instance of X in version 1.

5.5 Learning Outcomes

During our work on the catalogue, we were able to identify a selection of recurring problems that must be solved when implementing round-trip migrations. In the following, we discuss some of these problems by reformulating them independently from the concrete cases in which they appear.

Using Default Values

Many of the represented migration strategies leverage the use of default values. This is usually required when the target model version requires some sort of information to be available, while the source model version allows for the omission of said information. In round-trip migrations this imposes a challenge: When migrating back to the original model version, we must be able to detect default values so that we can avoid to introduce redundant information into the original model instance. Furthermore, our criteria for successful RTMs without modification require that default values do not appear in the round-trip migrated instance (cf. $g(f(m)) = m$ Def. 2.7). To address this issue, we recommend the use of modification detection in order to detect unmodified default values and replace them with their original representation (e.g. an absent optional field). In case a modification introduces instance data that is equal to the corresponding default values, we assume an explicit user intent and therefore migrate such changes back to the original model version. For these cases, the criteria for successful RTMs with modification apply. Therefore, using default values in migrations always entails the need for modification detection in order to distinguish explicit user intent from implicitly-set default values.

Information Redundancy

Redundancy of information imposes another challenge. More specifically, let us consider an original model version that models a certain bit of information using a single construct (e.g. a single field). Let us further consider another model version in which this bit of information is to be found in two different places (e.g. two separate fields). Informally, a functional dependency exists between these two sites and modifications must always equally apply to both sites. However, in the concrete case this may not always hold true. Therefore, the implementation of a migration usually encodes a precedence between the two redundant sites of information. As a consequence, if a modification updates the information inconsistently, a round-trip migration may dismiss the change and prioritize differently. While in data modeling, redundancy is usually undesired, it may still exist and it is important to note this implication for round-trip migrations.

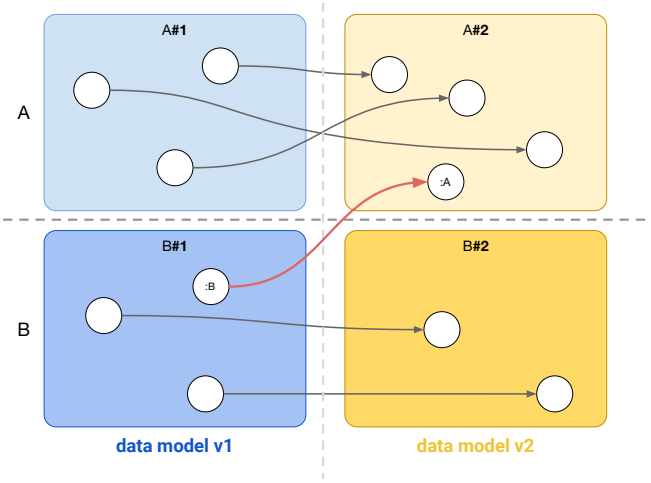


Figure 5.2: Schematic illustration of type A and B in different versions with corresponding representative objects in their instance sets (white nodes). The arrows demonstrate the mapping of objects that is imposed by a translation layer. Most objects are migrated within type boundaries (e.g. object of type A#1 to object of type A#2). The marked objects `:B` and `:A` however, are migrated across type boundaries. In these cases, the same system entity is represented as B in model version 1 and as A in model version 2.

Migration Across Type Boundaries

Finally, another recurring challenge is the migration across type boundaries. In some cases, it is required to migrate objects of a type A of the first model version to a type B of the second model version. This is to be differentiated from a simple type renaming, since we further assume that the types A and B exist in both of the model versions. However, some of the objects in the corresponding instance sets of A and B conceptually need to be moved to the other set during migration: The objects are migrated across type boundaries. An illustration of this idea is given in Figure 5.2. In migrations that face this issue, we must differentiate between objects that originally stem from the same type (e.g. B#1 to B#2) and objects that originally stem from a different type (e.g. A#1 to B#2). In order to fulfill our criteria of successful RTMs in these cases, we have found the use of trace links and runtime type checks to be an effective measure. More specifically, using trace links and runtime type checks we can differentiate the two types of objects and apply according migration strategies.

6 Case Study

6.1 Motivation

In this section we present a case study that we conducted to evaluate the results of this thesis in three main regards: (1) We evaluate how the presented framework for executing round-trip migrations can be applied to the concrete case of a real-world data model. (2) We evaluate the coverage and effectiveness of our scenario catalogue. More specifically, we determine whether the discussed scenarios appear in real-world models and vice-versa. Furthermore, we re-evaluate whether the various assumptions made in the scenario catalogue, similarly apply to the concrete case. And lastly, (3) we examine how feasible our requirements of successful round-trip migrations (with modifications) are for a real-world data model.

6.2 Overview

On an abstract level, our efforts for this case study can be structured into the following consecutive steps:

1. At first, we examined the main subject of the case study: A real-world data model that originates from an e-commerce web application. The application is still in active use and development. Therefore, the data model undergoes constant evolution. See section 6.3 for an overview of the data model.

2. In a second step, we examined the changes that were made to the model during a time period of 3 months. Since all changes to the data model were captured by a version control system, we were able to fully reconstruct an old version of the model. Using the current and the reconstructed old version, we extracted two snapshots to use in the case study. We will consider the reconstructed old version, the model version 1, and the version at the time of writing, the model version 2. Based on these two sources, we translated the original data model sources into an equivalent N4IDL representation using the versioned type syntax of N4IDL (see 3.2 Versioned Types).

3. Based on the first N4IDL implementation of the data model, we identified 12 concrete changes that could be observed between the two model versions (the two snapshots). Using our scenario catalogue, we related the observed changes to the discussed scenarios and designed a first migration strategy. For a further classification of the observed changes, see 6.3. Based on the initial design, we implemented a translation layer using N4IDL migrations. The implementation of the migrations is discussed in section 6.4.

4. To evaluate the fully implemented translation layer, we used randomly generated instance data to perform a large number of migrations. We checked each executed round-trip migration for our defined requirements of a successful RTM. For the case of RTMs with modification, we also generated random modifications of the data model instances and used a transformational approach to map those modifications back to the other model version. Using this test setup, we were able to check each executed RTM with modification for our defined requirements of successful RTMs with modification. For more details on this test setup, see section 6.5. The results of these different test runs are presented in section 6.6.

5. Finally, we reflected on the experience of implementing a translation layer from a developer's perspective. A discussion of the perceived effectiveness of our framework, as well as the general complexity the concept of round-trip migrations holds, is part of the conclusion of this case study in section 6.7.

6.3 The Data Model

The data model which forms the subject of this study, is used by a real-world e-commerce web application. Some of the modelled domain entities include products, orders, search queries, table reservations and various different types of quantities (e.g of monetary or metric nature). The subset of type declarations extracted for this case study includes 86 classes and 22 enum types. The extracted set of types forms a self-contained subset in the overall model. Furthermore, some minor editing was required to make the original sources suitable for this case study. The original implementation of the data model was given in the general purpose programming language Eclipse N4JS [1]. However, since N4IDL is derived from N4JS, the translation of the data model could be highly automated (e.g. insertion of version declarations).

The two extracted versions of the data model exhibited 12 distinguishable changes on the data model level. All of the observed model changes could be related to at least one of the discussed scenarios of our scenario catalogue. A full classification of the observed changes is given in table 6.1.

Model Change	Description	Number of Occurrences	Related Scenarios
Rename Field	The name of a field is changed.	4	1
Generalize Field Multiplicity from 0..1 to 1	The multiplicity of a field changes from 0..1 (optional) to 1 (mandatory).	2	8
Generalize Multiplicity 0..n to 1	The multiplicity of a field changes from 0..n (array) to 1 (mandatory).	1	10
Add field (functionally independent)	A new field is introduced. The new field does not have any functional dependencies on existing fields.	3	2, 4
Change super type	The super type of a class is changed.	1	6
Change type of a field	The type of a field is altered.	1	7

Table 6.1: A classification of all observed data model changes during a development time of 3 months.

6.4 Implementing Migrations

Given the observed changes as well as the insights gained from the scenario catalogue, an N4IDL translation layer was implemented. As apparent from the previous section, the number of total changes is much smaller than the number of classes in the data model. As a consequence, there are many types which remain unchanged from model version 1 to model version 2. For objects of these types, it suffices to migrate by copying all available field values. In other cases however, it is required to implement custom N4IDL migrations to accommodate for model changes. In the following we will discuss how both of these cases are handled by the implemented translation layer.

For objects of types which remain unchanged between the two model versions, a common migration strategy may be applied, which handles such cases using a generic approach. Since an object of version 1 of such a type is structurally equivalent to its counterpart of version 2, a simple copying of its field values suffices to fulfil the basic requirements of a successful round-trip migration. However, for all fields of that object that reference other objects of different type, we must ensure that the referenced objects are not just copied over but migrated using an appropriate migration. Therefore, the basic strategy is to copy over all primitively typed field values and to dispatch migration calls for all other referenced object. An example of such a *migration-by-copying* strategy is illustrated in Figure 6.1.

In the implemented translation layer, a generic migration that applies a migration-by-copying strategy forms the foundation of the translation. Per default, an object is migrated using this generic migration strategy, except for the cases where a type-specific migration is declared. In the context of N4IDL's type-dependent dispatching of migration calls, this is implemented in terms of a migration that is declared for the implicit common super type of all model types: `N40bject`. Instances of the implicit-common-super-type concept can also be found in other languages such as the `0bject` type in Java [16].

For the case study, a total of 15 custom N4IDL migrations was implemented. The custom migration code totalled to 191 lines of code (without empty lines and comments). The set of custom migrations was declared for 7 different types, which implies at least two migrations per type to round-trip migrate instances in both directions. The observed model changes often occurred as isolated atomic change. However, we also observed the overlapping of multiple changes for a single field. For instance, in one case both the type as well as the name of a field changed from one model version to the other. For isolated atomic changes, the implementation of migrations was mostly lead by re-using exemplary snippets from the scenario catalogue. In the case of overlapping changes, the implementation of migrations tended to be more complex and required additional attention.

In some cases, the inclusion of traceability information was required to guarantee a successful round-trip migration of data model instances. The implemented translation layer makes use of both N4IDL traceability

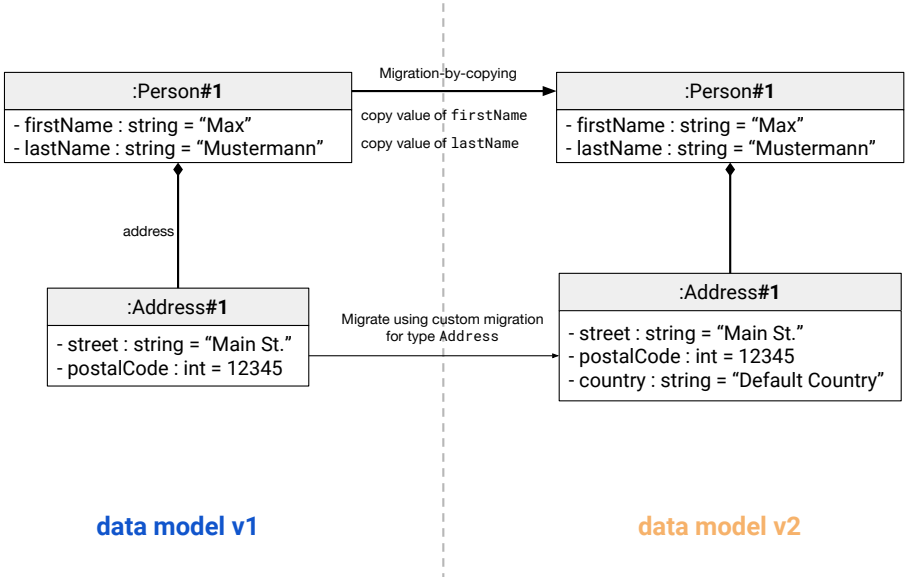


Figure 6.1: The object of type `Person` can be migrated using a generic migration-by-copying strategy. The referenced object of type `Address` however, must be migrated using a custom implementation, due to changes on the model level between version 1 and 2 (addition of field `country`).

features, trace links and modification detection. In general, according to subjective developer experiences, the use of modification detection often entailed a much higher complexity as it was harder to reason about all side-effects. Compared to that, the use of trace links was often perceived as rather intuitive.

6.5 Testing the Implemented Translation Layer

To verify the correctness of the implemented translation layer in terms of our defined requirements of successful RTMs, we performed a large number of migrations using it ($\sim$ 400 000 migrations). In order to achieve a high diversity in the set of migrated instance, we used a random instance generator which allowed us to generate a very large amount of instance data of high variety and arbitrary complexity (e.g. number of objects per instance, number of cycles in the object graph). For the case of RTMs with modification, we chose a similar strategy by additionally applying randomly generated instance modifications to the migrated instance. In both testing scenarios, we perform a round-trip migration and then check for the criteria of a successful RTM (with modification).

The random instance generators in use were specifically implemented for N4IDL and are designed to produce a random stream of N4IDL instances that fully exhaust the set of specification concepts N4IDL supports (e.g. various different field multiplicities, null-values, cyclic structures, etc.). In our initial terminology of data model semantics (cf. Definition 2.1), we may think of those instance generators as an approximation of drawing a random instance from the set of all perceivable instances (semantics) with uniform probability. However, due to the infinite nature of most data model semantics, we must keep in mind that the instance generators only represent an approximation of such a uniformly random selection. In the bigger picture however, the assumption remains that a large number of migrations of randomly generated instance data covers most relevant corner cases. Therefore, we see this testing setup as a limited validation of the translation layer's correctness.

For the case of *round-trip migrations without modification*, the assertion for each migration of random instance data is that we do not observe any loss of information. More specifically, we check that the round-trip migrated instances equal the original instances and thus all information has been preserved.

For the case of *round-trip migrations with modifications* our test setup is more complicated. In order to reach a high coverage of potential corner cases, we must additionally generate random instance modifications of the migrated instances. However, to check for the success of such a RTM with modification, we must also be able to translate the random modification of the migrated instance back into the original model version. For an illustration of this concept, see Figure 6.2. To enable a translation of modifications, we applied a model-based approach. We first designed a modification model which allowed us to represent any potential modification of a data model instance (e.g. change field value, set field to null, insert array element, delete array element). We then implemented model transformations that translate a modification of one model version to a modification

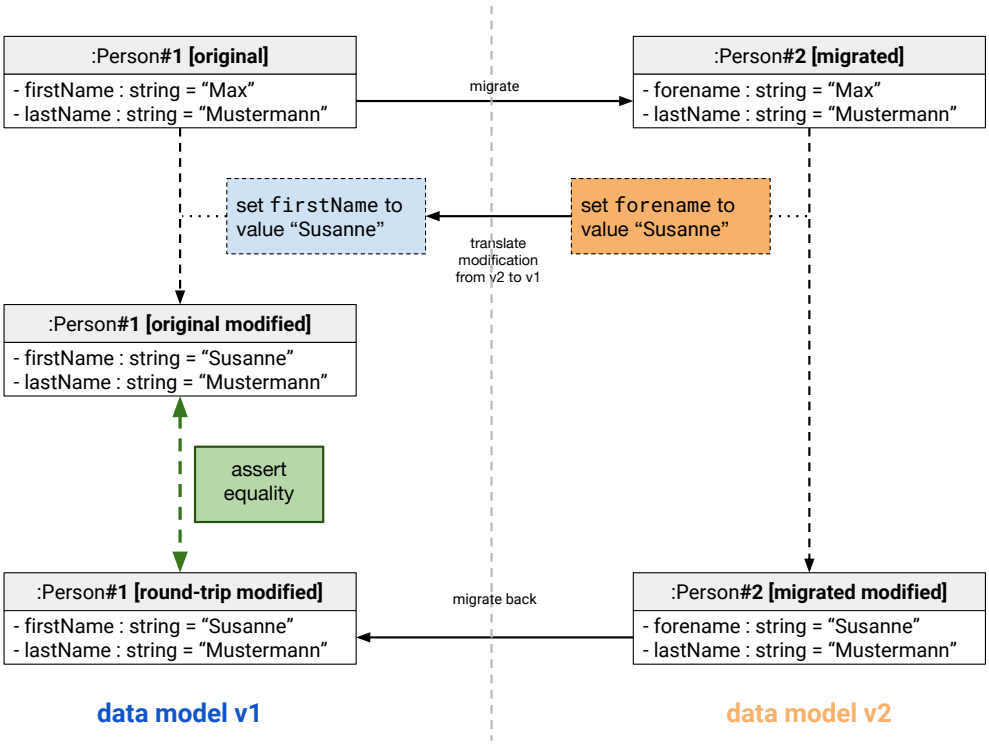


Figure 6.2: To check for a successful RTM with modification, the modification (set `forename` to value "Susanne") of the migrated instance must be translated to model version 1 (set `firstName` to value "Susanne"). Given such a translation of modifications, we may assert equality of the original modified instance and the round-trip migrated modified instance.

of the other model version. In Figure 6.2 this would be a transformation that transforms a change-field-value modification of field `forename` to a change-field-value modification of field `firstName` in model version 1.

We can also relate the concept of translating modifications from one model version to the other, to our initial definition of successful round-trip migrations with modification (cf. Definition 2.10). In our definition we define c_1 and c_2 to be equivalent modifications in the two versions of the data model, in that they represent the same system change c. In case of our test setup, we randomly generate one of c_1 and c_2 and use our modification transformation to map the generated modification to its counterpart of the other model version. Thus, our modification model transformations provides a bidirectional mapping between c_1 and c_2.

By executing a large number of migrations with these two test setups for RTMs with and without modification, we aim at assuring a certain robustness and limited correctness of the translation layer with regard to our definition of successful round-trip migrations.

6.6 Results

In this section we present the overall results of the migration of a large number of random instances as described in the previous section.

Round-Trip Migration without Modification

We executed 100 000 round-trip migrations (without modification) both from model version 1 via model version 2 and vice-versa. The number of objects per instance reached a maximum of about 1600 objects in one instance and averaged out at about 32 object per migrated instance. During that, we used each type of the data model as the migration root for about 980 different executions. All executed migrations fulfilled the requirements of a successful round-trip migration without modification.

Round-Trip Migration with Modification

Similarly, we executed a set of 100 000 round-trip migrations with modification of both model version 1 via model version 2 and vice-versa. Since the same instance generators were used, the random instance data exposed similar characteristics as in our tests of RTMs without modification. Concerning the randomly generated modifications of the migrated instances, we observed a maximum of 1230 atomic modifications in one instance (e.g. change field value, set field to null, insert array element) and an average value of about 9 modifications per execution. In all migrations, the translation of modifications from one model version to the other and the consecutive application of the translated modification as illustrated in Figure 6.2, always yielded equal *original modified* and *round-trip modified* instances. Therefore, we may assume that in all executions, the RTM with modification was successful.

To summarize, we ran an overall of 400 000 unique migrations with random instance data using the translation layer we implemented for the data model of this case study. For all of the migrations we could confirm that they were successful RTMs according to our definition. There was not one instance of a migration were the translation layer did not fulfil the requirements of a successful round-trip migration. Based on that, we assume a de facto correctness of the implemented translation layer.

6.7 Conclusion

In this case study, we examined a real-world data model with regard to our concept of round-trip migrations and the N4IDL migrations framework. After identifying the model changes that were observable during a development time of 3 months, we were able to relate all changes to at least one scenario in our catalogue. We interpret the fact that all observed changes were discussed in our catalogue as a good sign for its thoroughness. While some observed real-world model changes were clearly assignable to one of our scenarios, others could be classified as a combination of multiple discussed scenarios. In all cases, the catalogue formed an important knowledge base for the implementation of correct and effective real-world migrations.

Based on our tests with random instance data, we could further our confidence in the migration strategies proposed by the scenario catalogue. More specifically, we were able to confirm that the proposed strategies did not violate our central requirement of preventing loss of information during round-trips.

From a developer's standpoint, our proposed framework for the specification of data models and the implementation of migrations (N4IDL) proved to be an effective tool. We did not face any conceptual issues during the realization of a real-world round-trip-migrating translation layer. Specifically the type-dependent dynamic dispatch of migrate-calls fit the requirements of a generic fall-back migration very well (cf. migration-by-copying as discussed in section 6.4).

In some cases, the implementation of correct migrations did expose unexpected complexities. This was mostly the case for migrations which had to handle multiple overlapping model changes (cf. Section 6.4). While our catalogue does provide a knowledge base on how to handle isolated model changes, for overlapping changes, one must expect additional development time due to the increased complexity. Furthermore, the implementation of a migration significantly gains complexity, with an increasing use of traceability information. According to subjective developer experiences, the use of trace links may be regarded as more comprehensible compared to the integration of modification-related traceability.

Finally, to conclude this case study, we may use our results to address our third research question: *How feasible are the requirements of successful round-trip migrations for a real-world data model?* In general, the successful implementation of a translation layer for this case study exhibits a real-world example of a working translation layer that fulfils our RTM criteria. However, this case study must be seen in relation to the number of model changes (12) that were handled by the translation layer. One may argue, that the number of changes is comparatively low when considering larger data models and longer time periods of change. Therefore, we expect that the concept of successful round-trip migrations has its limits when it comes to the magnitude of change. More specifically, we assume that for very large data models, the implementation of round-trip-migrating translation layers implies a considerable implementation effort with regard to the migrations and may not be feasible.

7 Related Work

In this section we will discuss the concept of round-trip migrations and our proposed framework in the context of related work. Generally, our problem domain touches on a wide range of different fields of research. For instance, *schema*, *grammar*, *format* or *meta-model (co-)evolution* are all examples of areas in which the evolution of a specification formalism is studied. However, in the following we mainly focus on the related fields of *schema evolution* and *view update translation* in database systems as well as *metamodel co-evolution* in the more general modelling domain. Starting from the early years of the 1980s, a large body of related work has been published in database systems. By discussing it here, this thesis is compared to well-established research that is widely applied in practice. Furthermore, we focus on the field of metamodel co-evolution, as in past years it has seen a high interest from the research community. Therefore, the field seems promising to us in order to position our work in the midst of more recent approaches. Finally, we will relate the implementation of *traceability* in our framework, to approaches in *requirements* and *model-driven engineering*.

7.1 Database Systems

7.1.1 Schema Evolution

In database systems, the research topic of *schema evolution* concerns the modification of the schema of a populated database. According to Roddick [31], ”schema evolution is accommodated when a database system facilitates the modification of the database schema without loss of existing data”. Based on the nature of schema change, existing data must potentially be migrated to restore consistency with the modified schema.

The concept of *schema versioning* represents an even stricter view on schema changes. In comparison to schema evolution, schema versioning further implies the ability to access the stored data via arbitrary versions of a schema [31]. This is also described by the terms of backwards and forward compatibility as outlined in [29]. A database system is said to be *backward compatible*, if applications that were linked against the current version of a schema, can access data that was created under the previous version of the schema. *Forward compatibility* on the other hand, is accommodated when the database systems allows applications that were linked against an older version of a schema to work with data that has been created under a newer version of the schema. According to our initial definition of successful round-trip migrations, we require both forward and backwards compatibility. Therefore, we may say that only database systems that implement schema versioning provide similar functionality to our presented framework for round-trip migration.

In the past, various different implementations of schema evolution and limited schema versioning have been proposed. Some publications even specifically apply to the field of object-oriented database systems (OODBMS). In OODBMSs, a scheme can intuitively be understood as the declaration of an object-oriented type/class. Therefore, in the following, the object-oriented schema evolution parallels the evolution of our data models as defined in the beginning of this thesis (cf. section 2).

One prominent example of an OODBMS with support for schema evolution is the ORION database system as introduced in [5]. For ORION, Banerjee et al. propose an algebra of primitive operators for schema evolution. The authors proof the completeness of their change algebra in that it can be used to model all possible schema changes. Additionally, they define rules and invariants that further restrict the set of possible changes. In the original version of ORION [5], the authors intentionally define a set of invariants, that allow them to entirely avoid the actual migration of instance data. The schema evolution is limited in such, that all required changes on an instance level can be performed on-demand when instances are returned as result of a query (e.g. hiding of deleted fields, setting of default values, etc.). However, at no point are those adaptations persisted or applied to the physically stored instance data.

In later publications, this operator-based approach is extended by the support for automatic and user-defined migration functions for instance data. With the SERF framework [7] for example, Claypool et al. demonstrate a framework in which the change of the schema and a corresponding update of the instance data is always coupled. Based on this concept, they define an extensible set of complex operators that can be re-used by developers to implement non-trivial domain-specific schema and instance modifications.

In our framework, the overall setup of evolution is inherently different from this. Since we assume the two versions of a data model as given, we cannot track the concrete changes that lead to the differences between the versions. Therefore, an operator-based approach is not directly applicable to our use case. Instead, we

outsource the actual implementation of migrations to the developer. Furthermore, neither of these systems implement actual schema *versioning* since they mutate the schema directly and do not keep a copy of the older schema version. Nonetheless, with our function-based implementation of migrations, we aim at a similar idea of modularity and re-use of migration strategies. In our framework, common strategies can be externalized into generic helper functions, which can in turn serve as building-blocks for more complex migrations.

Apart from such operator-driven approaches, others propose systems in which the schema change is handled independently from the instance data migrations. For instance, the database systems O_2 [11] and OTGen [27] both provide their own languages for the definition of migrations. This generally allows arbitrary schema evolution while it is the responsibility of the user to migrate instance data correctly, if even possible. However, both of these systems implement mere *schema evolution* without any support for actual versioning in terms of forward compatibility.

Finally, one approach that is of special interest to us, is CLOSQL as proposed by Monk and Sommerville in [29]. They provide support for schema versioning by the implementation of so-called *update* and *backdate* functions. These functions are used to migrate instance data between the different versions of a database schema. In comparison to other approaches, they focus on the implementation of *schema versioning* instead of *schema evolution*. A change of the schema actually constitutes the creation of a new schema version instead of the modification of the current schema. By that, they implement *schema versioning*, as defined above, since the system may be used to query data instances using arbitrary versions of the schema. As in our proposed framework, they note that the ”end-user has to define his/her own update and backdate methods” [29]. Therefore, similar to our approach, they outsource the complexity of the actual instance migration to the developer. CLOSQL even implements limited support for mechanisms that we would consider traceability features. As Monk and Sommerville outline in [29], CLOSQL has the ability to store values during the execution of up- and backdate functions that would otherwise get lost. In later migrations, the system provides a storage that can first be queried for existing values to restore. However, it is not addressed how exactly the implementation of update and backdate is supported and to what extend a re-use of complex migration code is possible.

To conclude, the field of schema evolution discusses many problems that are similar to those we face with round-trip migrations. Especially, the idea of schema *versioning* seems promising. For future work it will be of interest to investigate a further knowledge transfer with regard to automating instance migration. Many of the discussed systems however, do not provide the forward and backwards compatibility that is implemented with our framework. Specifically the use of traceability information is not a matter of interest, since a popular use case of schema evolution, is to upgrade instance data from some ”old” version to a ”newer” version. This unidirectional character inherently differs from our idea of continuously round-tripping between two versions.

7.1.2 View Update Translation

Many database systems provide the feature of database views. In general this encompasses the idea of inferring new virtual sub-schemata using queries that operate on the base schemata of a database. By that, users gain the ability to define aggregating, summarizing or limiting views on the stored instance data. This feature can also be used, to ensure backwards compatibility by re-defining a previous schema version as view in the current database. The field of *view update translation* addresses the problem of performing updates on database views. More specifically, various publications propose methods for determining the required changes to the actual underlying instance data based on user-issued updates on a database view. This closely corresponds to our concept of round-trip migrations with modification, where we seek a reasonable mapping between the modifications c_1 and c_2 in the corresponding data model versions (cf. Def. 2.9). Our scenario catalogue discusses many concrete instances of this problem.

In [8], Dayal and Bernstein propose an initial formalization of the update translation problem. In their work, they describe the concept of *clean sources* and regular *sources*. A result tuple of a database view is said to be of clean source if and only if a one-to-one relationship exists between it and a tuple in the original instance data. Based on that, they propose a definition of correct update translation that is focused on the idea of preventing unintended side-effects. By their understanding, a view update can only be *performed exactly*, if there exists an update of the underlying instance data that has no other side-effect than the specified modification. In more recent work, such as [12], this idea of a precise update translation is also adopted. This core requirement is comparable to our definition of successful round-trip migrations with modification, since we also assert equality of the original and round-trip migrated instances after applying c_1 and c_2 respectively (cf. Def. 2.9).

Based on the foundational work by Dayal and Bernstein [8], further approaches emerged. In [4], Bancilhon et al.

present a method for update translation ”under *constant complement*”. Views are always defined together with a complement, which extends them by the required data to reconstruct instance data that conforms to the original base schema. As a consequence, updates to the view in combination with the fixed complement translate to updates of the original instance data. In our domain of object-oriented data models, this corresponds to a migration strategy in which we choose static default values for all missing features without acknowledging any context information. As we have seen (cf. 2.3Conceptual Limits), such a strategy is not feasible for our case.

With this thesis, we reside closer to the concept of *dynamic views* as presented in [18]. According to Gottlob et al., a dynamic view is a view definition together with a (conditional) update policy. The policy specifies how view updates are carried out on the original instance data [18]. This is comparable with our concept of instance migrations. Gottlob et al. further provide a classification of *consistent views* which guarantee an unambiguously determinable update strategy. It remains to be addressed by future work, how such a classification translates to our problem domain and how it affects the class of model differences that can successfully be round-trip migrated.

In recent work, the idea of dynamic views is further complemented by the use of bidirectional transformation languages [12] [28]. With those, the authors intend to facilitate the complex development process of a view definition and a correct update policy. By using bidirectional view definition languages, the update policy can automatically be inferred from the view definition. In comparison to that, our framework offers more flexibility by letting users implement unidirectional migrations imperatively. However, in future work the integration of a bidirectional migration language could be worth investigating.

While the discussed approaches so far were initially conceived with a relational database scheme in mind, their results conceptually translate to our problem domain of object-oriented data models. Nonetheless, other authors specifically address the problem of view update translation for object-oriented database systems. In both, the O_2 database system [2] and the description of *updatable views* by Scholl et al. in [32], the problem is addressed by defining custom view definition languages. By that, the set of possible projections is limited in such, that the ability to automatically infer an update policy is maintained. As a consequence, view definitions represent bidirectional transformations. However, since in our framework we allow arbitrary differences between data model versions, the applicability of such an approach is limited.

7.2 Metamodel Co-Evolution

In recent years, model-driven software engineering has become increasingly common practice for real-world scenarios. At the center of it, the concept of metamodels allows for the formal specification of models which a tailor-made for a specific domain. Metamodels constitute an artifact of the overall software development process and therefore undergo evolution. The field of metamodel co-evolution asks the question: *Based on metamodel evolution, how can we co-evolve models to maintain their conformance with the metamodel?* The relationship between metamodels and models is defined as a specification formalism which is comparable to the relationship between our concept of data models and data model instances. Therefore, many of the results in the field of metamodel co-evolution are also applicable to our problem domain of round-trip migrations.

A large body of work in the field of metamodel co-evolution discusses the automatic generation of migration (co-evolution) strategies for models, based on metamodel changes. While in our approach, we do, as of now, not support the automatic generation of instance migration strategies, we do provide a framework that allows to re-use pre-defined migration strategies in terms of helper functions. As we will see later on, this could already be considered a semi-automatic approach to co-evolution. Nonetheless, we think that the automatic generation of instance migrations based on data model changes represents a great research opportunity for future work. Finally, it remains to note, that one important difference between the problem of round-trip migrations and metamodel co-evolution is the direction of change. Generally, metamodel co-evolution considers a one-way evolution of the metamodel (towards the ”newer”, evolved version). Therefore, traceability throughout multiple steps of (co-)evolution is usually not a matter of interest. Neither is the idea of switching between two metamodel versions as we do in the scenario of round-trip migrations.

A recent survey in the field [19], identifies three main steps that are common to most co-evolution approaches. In a first step, (1) the metamodel changes are collected. On the one hand, this is done using a difference analysis between two given versions of the metamodel (e.g. Cicchetti et al. in [6]). Other approaches, propose an operator-driven method (cf. Schema Evolution), that allow for a change detection based on the explicit changes a user performs (cf. COPE in [20]). In a second step, (2) the metamodel changes are identified. That is, based on the collected list of changes, atomic and complex changes are classified. Generally, the better the

identification of changes, the likelier is the generation of a correct co-evolution strategy [19]. In the last step, (3) the models are resolved. Compared to our framework, this corresponds to the execution of instance migrations. However, in metamodel co-evolution, this step also implies the (possibly automatic) generation of a co-evolution strategy.

In [19], Hebig et al. observe three *benefit classes* with common approaches:

- *1:1* User intervention is required for every model resolution. That is, a developer must assist the migration of every single model. For our use case, this is not feasible since, as we outlined in our initial motivation, we want to be able to migrate instance data on-the-fly in highly connected systems.
- *1:n* User intervention is required once for every metamodel change that is detected. More specifically, a developer specifies essential parameters or details of a migration strategy that can be applied for all models that are concerned by the change. In our framework we implement this by requiring the implementation of migrations, that can automatically be executed at runtime.
- *0:n* The system does not require any user intervention. This can only be implemented, by addressing every metamodel change with a generic default co-evolution strategy. As is noted in [19], in this case, results are "not guaranteed to be the one desired for the concrete models at hand". In many cases, a generic resolution strategy does not apply to all concrete instances of a specific metamodel change.

Most publications propose a system that can be assigned to multiple benefit classes. In [6] for instance, Cicchetti et al. propose the differentiation of *resolvable* and *non-resolvable* changes. For the former, automatic resolution strategies can be provided. Non-resolvable changes usually require user input to integrate domain knowledge with the migration strategy. An instance of a *0:n* concept, is constraint based model search (e.g. Demuth et al. in [9]). Rather than examining the actual metamodel change, the authors aim to co-evolve models by performing a constraint-based search to restore conformance with the changed metamodel.

With our framework for round-trip migrations, we implement benefit class *1:n*. At the time of data model evolution, the developer provides additional information in terms of the migration implementation. At migration time, those can be used to automate the instance migration without requiring any additional user intervention.

Many of the presented methods for metamodel co-evolution in [19] make use of a difference-based change collection (see above). However, other publications propose operator-driven methods. Instances of such can be found with the COPE project [20] or with a method for evolving *domain specific languages (DSL)* as presented in [34]. They are comparable with the idea of operator-driven schema evolution in database systems, as discussed in a previous section (cf. section 7.1.1). In fact, the COPE project was strongly inspired by the SERF framework [7] [20]. Such operator-driven methods usually go along with editors that limit the editing process by designated UI concepts (cf. categorized as UI-intrusive approaches in [19]). Hebig et al. argue in [19], that operator-based systems can oftentimes be difficult to use, since they usually require a high-proficiency with regard to the set of available operators. The authors of COPE on the other hand, intent to decrease the overall migration effort by this large number of predefined migration strategies [20]. While not operator-driven, we position our framework close to these projects, since we also encourage modularity and re-use on a language level.

To conclude, we may say, that although metamodel co-evolution is usually performed on a different level of abstract (metamodel/model vs. model/instance), many concepts can be transferred to our problem domain. In particular, the discussed benefit classes provide us with common ground that allows to position our rather novel approach in a wide field of recent research. Finally, the automatic generation of instance migration strategies based on data model changes, remains a promising direction for future work.

7.3 Traceability

In the past, various publications in the fields of requirements engineering (RE) and model-driven engineering (MDE) have formulated the need for traceability. In [33], Winkler and Pilgrim highlight different definitions of the term *traceability*: Firstly, they refer to the IEEE Standard Glossary of Software Engineering Terminology [22] which gives a broad definition of traceability as a "degree to which a relationship can be established between two or more products of the development process [...]" [22]. Alternatively, Gotel et al. understand by traceability the "ability to describe and follow the life of a requirement [...]". While both of these definitions are rooted in the field of requirements engineering, most research in MDE proposes definitions that evolve around the idea of traceability in terms of trace links between the in- and outputs of model transformations (cf. Paige in [30] and the Object Management Group (OMG) in [13]).

Although research in the fields of RE and MDE studies traceability of artifacts in a software development process, we may transfer many of the proposed concepts to our problem domain. With the integration of traceability, we aim at allowing the use of information on the life and history of an instance in further migrations. However, we must address the difference in abstraction between RE and MDE, and our approach. While both, RE and MDE usually deal with artifacts of the development process, we explicitly deal with runtime instance data. This constitutes an important difference in different regards. For instance, in RE and MDE, user intervention for the creation of traceability information is an option (cf. [33]). For the use case of our proposed framework, traceability must be a fully automatized feature, as user intervention at runtime is not feasible. Overall, we position our approach closer to the concept of traceability as it is implemented in model-driven engineering. In particular, MDEs strong focus on model transformation relates to our idea of instance migration.

One popular form of implementing traceability, particularly in MDE, is by the means of trace links. These links record a relationship between artifacts. Usually this can be across different levels or on the same level of abstraction. Examples include the relationship between a requirement description and the corresponding implementation or more generally the in- and outputs of a transformation. The semantics of a trace link is usually dependent on the concrete case and most publications propose their own traceability scheme (see [30] and [3] for examples). In [33], Winkler et al. compile a list of potential features a traceability scheme can provide. This includes but is not limited to the link cardinality, directionality and the availability of type information. As introduced in section 3.7.1, we implement trace links as unidirectional one-to-many links that document the relationship between data model instances and their previous revisions. For now, we do not support typed trace links, however, the statically typed foundation of the N4IDL language provides a strong incentive for future work in that direction.

Winkler and Pilgrim note in [33], that traceability, does generally not "imply a particular form or representation [...]". Therefore, although traceability in MDE often refers to the idea of trace links, traceability can also be accommodated by other means than links. For instance, an alternative form of traceability, is the detection of modifications of migrated instances that we provide in our framework.

Another important aspect of traceability information is its creation or recording. In general, it is desirable to automate this process. In requirements engineering however, this sometimes represents a challenge, especially in case of so-called *pre-requirements specifications* [17]. Such artifacts only exist in terms of informal descriptions and are harder to track as they may require manual efforts by the user. In MDE, this is usually less of a problem, as the heavy use of models mostly allows to produce traceability information as a side-product of transformations. These *on-line* approaches [33] have also been implemented for concrete model transformation languages, such as the Atlas Transformation Language (ATL) [23]. In that regard, we would like to highlight one publication: In [3], Amar et al. propose an approach to trace imperatively implemented model transformations. By the use of aspect-oriented programming [26], they separate the concern of implementing a transformation from the concern of recording traceability information. In our framework we propose a similar concept by automatically generating traces based on the call-hierarchy of migrations. While we do not operate on the instruction level, as it is the idea in aspect-oriented programming, we aim for the same separation of the implementation of migrations and the capturing of traceability information.

Finally, in the field of *model synchronization*, traceability finds an application that is very similar to ours. The general challenge of model synchronization is to keep multiple models that model different views on a system in sync, when at first only one of them is changed (e.g. the change of an entity name in one view, must propagate to all other views accordingly). In this context, we want to particularily highlight the work of Getir et. al in [15], in which the authors present a framework which allows to analyze past changes to different views and perform a correlation analysis on them. By that, they aim to gain a better understanding of the way changes in different

views correlate, so that later, a coupled evolution can be performed by suggessting users with parallel edit operations in related models (so-called co-evolution steps). While for our problem domain, it is not feasible to develop such an understanding over time, this work relates to ours in two ways: Firstly, for round-trip migrations we also must develop an understanding of how changes to the data model affect instance data. This can be seen as a synchronization problem between model and instances (cf. metamodel co-evolution). On a different level, we must also synchronize changes to instance data when dealing with round-trip migrations with modifications. Similar to Getir et. al, we exploit trace links in order to enable the succesful synchronization of instances of one model version with instances of another model version. It further remains to note that Getir et. al extended their work in [14] by an exemplary catalogue of such co-evolution steps for the concrete example of system architecture and fault tree models. With that they apply a similar approach as we did with our scenario catalogue.

To summarize, we have seen that clear parallels can be drawn between traceability, as it is implemented in RE and MDE, and our approach. In future work, we suggest an extension of our framework by the feature of typed trace links. More specifically, we would like to improve our language for migrations by statically typed trace links that guarantee the type of an obtained previous revision at compile-time. At the time of writing, we were not aware of any work that aims to transfer the concept of traceability to the instance level. However, the large fields of RE and MDE provided us with a good foundation for the design of our framework.

8 Conclusion

This section concludes this thesis. At first, we relate our results to our initial premise and summarize our contributions. In 8.2 we discuss some of the remaining questions while section 8.3 highlights a selection of future lines of research.

8.1 Summary

As the initial premise of this thesis, we considered version-heterogeneous distributed software systems. More specifically, we focused on the fact that distributed systems are often comprised of many components whose interoperation must be guaranteed. However, due to the use of many independent software components, this can be a challenge. One effective measure to attain consistency across such systems is the use of a common data model. Over time, data models need to be changed to accommodate for new requirements. Since it can be difficult to propagate such changes to all components of a distributed system, it is common practice to maintain multiple versions of a data model for different components. Therefore, to further guarantee the functioning of the system as a whole, it must be assured that the different versions in use, are compatible with each other. On the one hand, this may be achieved by assuring backwards compatibility across different versions. Yet, this can negatively affect the maintainability of the data model and generally impose constraints on any further development. With this thesis, we proposed a different approach: By the use of bidirectional translation layers between components of different version, the consistent functioning of the system is guaranteed, while allowing for non-backward-compatible data model changes.

This thesis first accomplishes a formal foundation required to specify the requirements of a bidirectional translation layer (cf. section 2). In an initial description of the problem, we introduced the term of *round-trip migrations* (RTMs) which captures the idea of translating the communication between two components of different version transparently. That is, without any adaptations, the two components are able to fully interoperate. A central requirement that we impose is the concept of *successful* round-trip migrations. In a second step, we furthered our understanding of RTMs by considering the modification of a migrated model instance, and formalized a requirement that encapsulates the idea of translating modifications transparently.

After developing a foundational understanding of the precise problem domain, we focused our efforts on the object-oriented modelling language N4IDL. With the motivation of allowing for the effective implementation of round-trip migrations, we developed a framework and execution environment that evolves around N4IDL as a modelling language (section 3). As core feature to allow for round-trip migrations, we implemented support for *traceability information* as known from model-driven engineering (cf. [33]). The framework is fully functional and provided with this thesis in terms of a reference implementation.

In order to evaluate the applicability of round-trip migrations to actual data models, we compiled a catalogue of round-trip migration scenarios. In a total of 21 scenarios, we discussed various different challenges that the implementation of round-trip migrations imposes. While the use of traceability information during the implementation of migrations can be rather complex, we see a lot of potential in the re-use of repetitive migration code. As a result of these efforts, we could identify many cases of non-trivial data model changes that can be successfully round-trip migrated using our framework. However, we also identified limits to the idea of round-trip migrations. Apart from the proposed framework, we see the scenario catalogue as the second main contribution of this thesis.

Round-trip migration scenarios allowed us to systematically approach many different cases of data model changes. As a contrast, we additionally carried out a a case study based on a real-world data model (section 6). In the case study, we implemented a fully functional translation layer using the proposed framework and the knowledge gained from the scenario catalogue. We used the implemented layer to round-trip migrate instances between two data model versions, which were based on snapshots in the change history of an e-commerce application. The bridged time period amounted to 3 months of active development time and included a total of 12 atomic data model changes. With this case study, we were successful in further verifying our framework, the scenario catalogue and the general concept of round-trip migrations.

8.2 Discussion

With this thesis, we propose a formal basis for the idea of round-trip migrations and provide a framework for their implementation. Generally, it remains to be seen how effective such an implementation can be applied in practice. Nonetheless, we already want to highlight three topics of discussion at this point:

The Limits of Round-Trip Migrations

As it became clear in our work on the scenario catalogue, a transparent translation between different data models is only feasible to a certain degree. Depending on the nature of transformation, a successful round-trip migration is not always possible. For instance the aggregation of instances into primitive values (cf. scenario 20) is generally non-injective and thus not invertible. The identification of a concrete class of data model changes which allow for round-trip migrations, imposes a significant challenge and is out of the scope of this thesis. However, we believe that our scenario catalogue can be a guide for the implementation of translation layers, that allow to exceed the extent of purely backward compatible data models. Therefore, we regard our proposed solution as an improvement over the conventional strategy that requires backward compatibility.

The Complexity of Implementing a Translation Layer

In our work on the case study and the catalogue, we were able to make first experiences with the implementation of round-trip migrations. Based on these experiences, we found the complexity of developing a translation layer to be rather high. Since migrations can be executed with a large variety of inputs as well as context information, their implementation can become a complicated undertaking. Especially, when considering that most migrations implemented in the context of this thesis were dealing with small exemplary data models. Therefore, we estimate the development effort required for the implementation of a correct translation layer as comparatively high.

We recommend that translation layers are made subject to a large number of extensive tests in order to ensure their correctness. In our case study we leveraged random instance generators to explore potential corner cases. However, random test data does not suffice for a complete testing strategy. Therefore, we additionally recommend manually written tests. The complementary use of visualization tools for selected round-trip scenarios has also proven to be of great help. In fact, all round-trip visualization of this thesis were generated with such tools.

Finally, a lot of the complexity involved in writing migrations is a question of the abstraction level. Our proposed framework provides migrations with a migration API on a comparatively low level. In future work (see below), we propose the implementation of a domain-specific migration language, in order to enable the implementation of migration strategies on a higher level. Our current approach encourages modularity, which allows the definition of such higher-level strategies. However, the provided set of generic migrations is limited to essentials for now. Therefore, we hope that future work in this direction will help to manage the complexity of implementing round-trip migrations.

The Usage of Translation Layers

In our initial premise, we propose a system that transparently translates instances between model versions. However, we must also consider effects of such a mechanism that go beyond the idea of successful round-trip migrations. For instance, translation layers may allow bad actors to gain read or write access to data that is otherwise protected. This could happen when a security measure is implemented with only one version of the model in mind. By migrating an instance to another version (possible via multiple other versions), access control specifications can be weakened or misinterpreted. Therefore, all new features (security or otherwise) that are added to an existing system, must also be tested against older model versions by considering translation layers. This can cause a significant development overhead. On the other hand, this must also be seen in comparison with the alternative of re-implementing the same feature for every model version separately.

Finally, our approach is yet to be evaluated from a performance standpoint. Both our core traceability features, trace links and modification detection can be very resource intensive. Therefore, it remains to be seen how they scale in large systems. For our reference implementation, performance was not a target criteria, thus, we suspect a lot of room for improvement in that regard.

8.3 Future Work

This thesis only represents an initial step towards a fully-functional stack for round-trip migrations. Thus, there are many open questions which need to be addressed by future work. In the following we will discuss a selection of three different lines of research which we regard as essential.

(Semi-)Automatic Generation of Migrations

In section 7 on related work, we discussed several publications that propose an approach to the automatic generation of migration strategies (e.g. [5] [6] [9]). The general consensus in both metamodel co-evolution and schema evolution seems to be, that a full automation is not feasible, due to the significance of domain knowledge for the concrete migration of instance data and models. However, many publications propose a hybrid approach in which parts or an initial migration strategy can be automatically inferred from model or schema changes. For future work, we propose to investigate further, how our framework can be extended to support a similar semi-automatized generation of migrations based on data model differences.

A Domain-Specific Language for the Implementation of Migrations

As noted above, the level of abstraction constitutes a significant source of complexity for the implementation of migrations. Although traceability is a language feature of the proposed framework, migration code still has to operate on a comparatively low level. In future work, it may thus be of interest to develop a domain-specific language for the implementation of migrations, which targets the currently provided low level migration API. For that, numerous related approaches exist in the field of model transformations (e.g. ATL [24]) or update translation (e.g. [12]). Specifically, bidirectional transformation languages seem to fit the concept of round-trip migrations very well.

Persistence of Traceability Information

Finally, future work must be done in order to address the topic of persisting traceability information. Specifically for system components which persist instance data (e.g. database systems), the captured traceability information must be persisted together with the instance. As opposed to scenarios in which traceability information can be considered volatile (e.g. on-demand translation of real-time communication), persisting components require traceability information to also be available when restoring an instance from storage. In order to solve this problem, a format for the storage of traceability information must be devised. As a side-effect, this may also yield performance optimizations with regard to the compression of traceability information.

References

[1] Eclipse N4JS, High-quality JavaScript development for large Node.js projects. http://www.eclipse.org/n4js/, accessed: 2018-03-18

[2] Abiteboul, S., Bonner, A.: Objects and views. In: ACM SIGMOD Record. vol. 20, pp. 238–247. ACM (1991)

[3] Amar, B., Leblanc, H., Coulette, B., Nebut, C.: Using aspect-oriented programming to trace imperative transformations. In: Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International. pp. 143–152. IEEE (2010)

[4] Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Transactions on Database Systems (TODS) 6(4), 557–575 (1981)

[5] Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. ACM SIGMOD international conference on Management of data 16(3) (1987)

[6] Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: Enterprise Distributed Object Computing Conference (EDOC), 2008 12th International IEEE. pp. 222–231. IEEE (2008)

[7] Claypool, K.T., Jin, J., Rundensteiner, E.A.: Serf: schema evolution through an extensible, re-usable and flexible framework. In: Proceedings of the seventh international conference on Information and knowledge management. pp. 314–321. ACM (1998)

[8] Dayal, U., Bernstein, P.A.: On the correct translation of update operations on relational views. ACM Transactions on Database Systems (TODS) 7(3), 381–416 (1982)

[9] Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Co-evolution of metamodels and models through consistent change propagation. In: ME@ MoDELS. pp. 14–21. Citeseer (2013)

[10] ECMA International: Standard ECMA-262 - ECMAScript Language Specification. 5.1 edn. (June 2011), http://www.ecma-international.org/publications/standards/Ecma-262.htm

[11] Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., Madec, J.: Schema and database evolution in the O_2 object database system. In: VLDB. vol. 95, pp. 170–181. Citeseer (1995)

[12] Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. ACM SIGPLAN Notices 40(1), 233–246 (2005)

[13] Frank, K.: A Proposal for an MDA Foundation Model. An ORMSC White Paper V00-02 ormsc/05-04-01. Object Management Group (OMG) (2005)

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

[15] Getir, S., Rindt, M., Kehrer, T.: A generic framework for analyzing model co-evolution. In: ME@ MoDELS. pp. 12–21 (2014)

[16] Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java language specification. Pearson Education (2014)

[17] Gotel, O.C., Finkelstein, C.: An analysis of the requirements traceability problem. In: Requirements Engineering, 1994., Proceedings of the First International Conference on. pp. 94–101. IEEE (1994)

[18] Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM Transactions on Database Systems (TODS) 13(4), 486–524 (1988)

[19] Hebig, R., Khelladi, D.E., Bendraou, R.: Approaches to co-evolution of metamodels and models: A survey. IEEE Transactions on Software Engineering 43(5), 396–414 (2017)

[20] Herrmannsdoerfer, M., Benz, S., Juergens, E.: Cope-automating coupled evolution of metamodels and models. In: European Conference on Object-Oriented Programming. pp. 52–76. Springer (2009)

[21] Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: International Conference on Software Language Engineering. pp. 163–182. Springer (2010)

[22] IEEE: Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990 (Dec 1990)

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

[24] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of computer programming 72(1-2), 31–39 (2008)

[25] Kehrer, T.: Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering. Ph.D. dissertation (2015)

[26] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: European conference on object-oriented programming. pp. 220–242. Springer (1997)

[27] Lerner, B.S., Habermann, A.N.: Beyond schema evolution to database reorganization. In: ACM SIGPLAN Notices. vol. 25, pp. 67–76. ACM (1990)

[28] Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: ACM SIGPLAN Notices. vol. 42, pp. 47–58. ACM (2007)

[29] Monk, S., Sommerville, I.: Schema evolution in OODBs using class versioning. ACM SIGMOD Record 22(3), 16–22 (1993)

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

[31] Roddick, J.F.: A survey of schema versioning issues for database systems. Information and Software Technology 37(7), 383–393 (1995)

[32] Scholl, M.H., Laasch, C., Tresch, M.: Updatable views in object-oriented databases. In: International Conference on Deductive and Object-Oriented Databases. pp. 189–207. Springer (1991)

[33] Winkler, S., von Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Software & Systems Modeling 9(4), 529–565 (2010)

[34] Wittern, H.: Determining the necessity of human intervention when migrating models of an evolved DSL. In: Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2013 17th IEEE International. pp. 209–218. IEEE (2013)

Appendices

A Inspecting the source code included with this thesis

The source code of this thesis can be accessed in two ways. On the one hand, we provide a fully configured virtual machine image that includes a modified version of the Eclipse N4JS [1] IDE. This allows for the inspection and editing of all bundled source code, in a feature-rich IDE based on the Eclipse Platform. On the other hand, the appended media also includes a raw representation of the source code in terms of the corresponding N4IDL and N4JS files. The raw representation may be imported into a compatible version of the Eclipse N4JS IDE, but also requires additional setup in order to configure the Node.js [3] based runtime environment. For readers who want to execute migrations and explore the features of N4IDL, we therefore recommend the use of our virtual machine image.

Virtual Machine Configuration

The virtual machine (VM) image is provided in terms of an Open Virtualization Format (OVF) 2.0 file. It is based on an installation of Xubuntu 18.04 and has been tested to run successfully with the following parameters.

Property	**Value**
Host	Windows PC with 3.3 GHz Intel Core i5; 8GB system memory using VirtualBox Version 5.2.10 on Windows 10 *or* MacBook Pro Late 2013; 2.3 GHz Intel Core i7; 16GB system memory using VirtualBox Version 5.1.26 on macOS 10.12
Guest System Memory	8 GB
Guest CPU count	4
Guest Video Memory	128 MB with enabled 3D acceleration

Note that the default keyboard layout of the virtual machine is set to German. You can switch to an American layout using the flag icon in the upper right corner of the user interface.

Furthermore, the virtual machine is configured for a single user account with name and password set to '`n4idl`'.

Virtual Machine Usage

In the running virtual machine, the IDE can be launched using the desktop shortcut *N4IDL IDE*. After startup, the IDE will present the user with a pre-configured workspace which contains the source code that is bundled with this thesis. The Eclipse Working Sets (top-level entries in the Project Explorer on the left) categorize the included projects (cf. Figure A.1a). *Runtime* contains all source code related to our reference implementation of an N4IDL migration runtime (section 4). *Catalogue* contains the project that represents our scenario catalogue in code (section 5). *Visualization* contains miscellaneous projects related to the visualization of round-trip migrations. Finally, *Other Projects* contains a *Playground* project which can be used to explore the features of N4IDL and our migration runtime (cf. Figure A.1b).

The project *Playground* contains two different N4IDL files `Playground.n4idl` and `PlaygroundWithContext.n4idl`. The playgrounds demonstrate the usage of the N4IDL migration runtime and allow for an easy adaption of the data models and migrations. The second playground `PlaygroundWithContext.n4idl` also demonstrate how to access context information in N4IDL migrations.

[3] Node.js: `https://nodejs.org/en/`

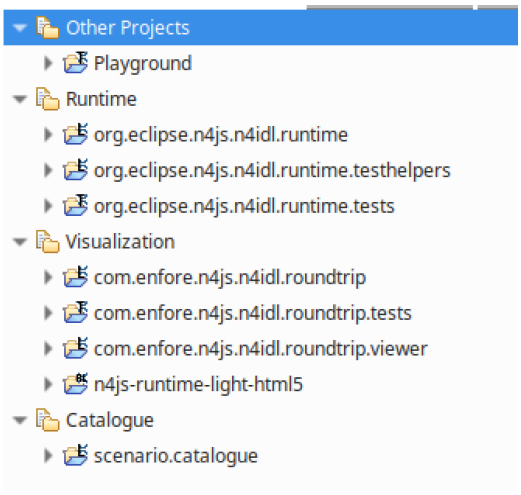


Figure A.1(a): An overview of all workspace projects in the virtual machine.

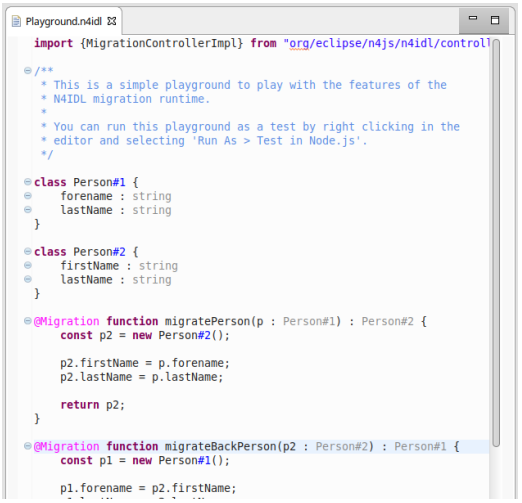


Figure A.1(b): The playground file that allows to explore the various features of N4IDL.

Raw Representation

As noted initially, the source code is also available in terms of a raw, textual representation. The raw sources can be found in the folder `Sources/` of the appended media. The folders in the `Sources/` folder correspond to the projects that are available in the virtual machine, excluding the miscellaneous projects with regard to the visualization of round-trip migrations.

B Inspecting the N4IDL migration runtime reference implementation

The sources of our reference implementation can be found in the `org.eclipse.n4js.n4idl.runtime` project. The file `src/org/eclipse/n4js/n4idl/controller/MigrationController.n4js` represents the entry point to our implementation of the `MigrationController` interface (cf. section 3.8). Starting from this point, users may further inspect other implementation components. This includes tests which can be found in the project `org.eclipse.n4js.n4idl.runtime.tests`.

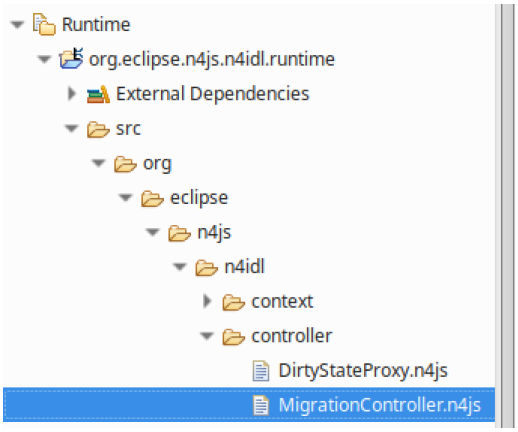


Figure B.1(a): The `MigrationController` implementation in the migration runtime project.

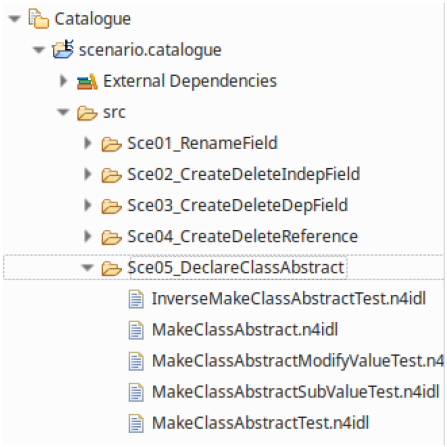


Figure C.1(a): The folder structure in the `scenario.catalogue` project.

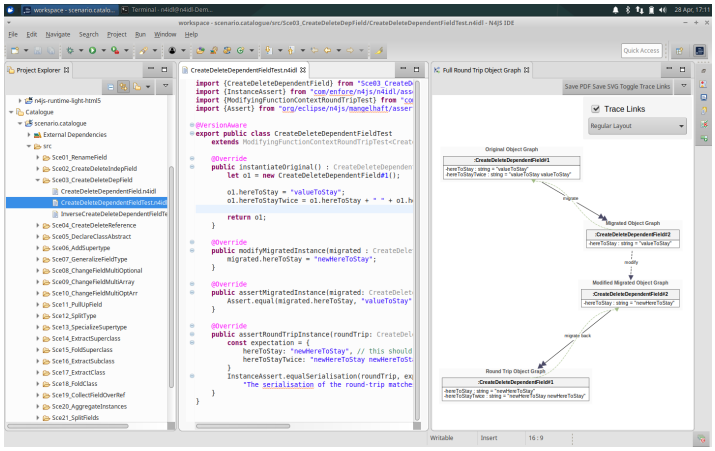


Figure C.1(b): A visualization of a round-trip migration scenario taken from our scenario catalogue.

C Inspecting the source code of the scenario catalogue

In the appended sources the scenario catalogue is represented by the `scenario.catalogue` project. The source folder of the project (`src/`) contains one sub-folder for each of our scenarios (cf. Figure C.1a). Each sub-folder contains the data model and migration declarations next to several files that represents test cases for the different variants in consideration (e.g. modifications, directions).

Furthermore, all included test cases can also be visualized. In order to do so, open one of the test files (e.g. `MakeClassAbstractModifyValueTest.n4idl`), right-click into the appearing editor and select *Run As* → *Launch with N4IDL RoundTrip Runner*. This will execute a round-trip migration and visualize the different round-trip stages in the *Full Round-Trip Object Graph View* on the right-hand side of the user interface (cf. Figure C.1b).

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 29.05.2018

Luca Beurer-Kellner