EMF-Syncer solution to TTC'20 round-trip migration case

Artur Boronat[1]

[1]School of Computing and Mathematical Sciences, University of Leicester, University Rd, LE1 7RH, Leicester, United Kingdom

Abstract

In this paper, we present a solution to the TTC'20 offline case *Round-trip migration of object-oriented data model instances* [1]. This case involves the application of maintenance tasks to web APIs that are associated with domain models so that updates are backward compatible. The solution presented in this article features the EMF-Syncer [2], a synchronization tool for bridging MDE-agnostic programs and MDE-aware programs, which may have some similarities in their object-oriented data models at run time. EMF-Syncer provides a generic synchronization strategy that exploits such similarities automatically and is, therefore, a suitable candidate for solving the proposed problems, since each problem relies on a small change and there is a large overlap between two versions of the same data model. In the paper, we used the case benchmark framework to justify that our solution exhibits a good balance between specification conciseness and performance.

Keywords

Model syncing, model-driven engineering, round-trip engineering

1. Introduction

The *Round-trip migration of object-oriented data model instances* case [1] for the TTC'20 exposes an evolution problem in the context of web API development, where different APIs work on top of a common data model. To accommodate new or changing requirements in the system, API designers need to ensure backward-compatible changes in the underlying data model. Assuming an agile software development environment where data models may evolve rapidly, the case proposes the use of round-trip migration services, relying on Model-Driven Engineering (MDE) technology, for enabling the co-existence of different versions of the data model at run time. In particular, the Eclipse Modeling Framework (EMF) [3] is used to encode data models with the EMF meta-modeling language, namely Ecore, so that tools built atop EMF can be used to implement data migration services.

The solution presented in this article features the EMF-Syncer [2], a synchronization tool for bridging MDE-agnostic programs and MDE-aware programs, which may have some similarities in their object-oriented data models[1], at run time. EMF-Syncer provides a generic synchronization strategy that exploits such similarities automatically and is, therefore, a suitable candidate for solving the proposed problems, since each problem relies on a small change and there is a large overlap between two versions of the same data model. The EMF-Syncer assumes that MDE-agnostic programs are implemented in a JVM language and that MDE-aware programs represent their data model using EMF-generated code.

Given a source data model (represented in a package of classes in the MDE-agnostic program), a target data model (represented as an Ecore model in an EMF-based program), and a collection of objects representing the source program state, EMF-Syncer translates the MDE-agnostic objects as an EMF model instance in the target EMF-based program, synchronizing the state of both programs at run time. EMF-Syncer automatically infers structural similarities between object-oriented data models by mapping object structural features by name, when found, translating both attribute and reference values. This translation can be performed using a push-based model, where the entire source program state is migrated, or using a pull-based model, where only those feature values accessed in the target program are migrated. Once synchronization is established, changes that have been applied to target EMF model instances can be incrementally back-propagated to the source MDE-agnostic counterpart. Incrementality of back-migration entails that only changes in target model instances are propagated back and merged within the source instance.

The aforementioned generic mapping strategy that is built in EMF-Syncer can be customized in order to allow for more complex data transformations between the data models involved. A domain-specific mapping strategy is declared with a mapping specification that maps a source feature type to a target feature type, possibly including feature value transformations, either from source to target, or from target to source, or both. Two main custom mapping strategies can be declared:

- Renaming of feature types: such renamings may affect the name of the class, where the feature is declared, and the name of the feature. This

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

artur.boronat@leicester.ac.uk (A. Boronat)
https://arturboronat.github.io/ (A. Boronat)
0000-0003-2024-1736 (A. Boronat)

[1]We refer to these data models as domain models in [2].

mapping strategy allows for transformations that modify the data model either syntactically, when either a class or a feature is renamed in the target model, or structurally, when a feature type is moved to an unrelated object type. In this case, feature values are transformed using the generic synchronization strategy.

- Transformation of feature values, with or without renaming of feature types: feature value transformations are expressed using Xtend lambda expressions defined for a contextual object type, from the source data model, and result in a single target feature value. This mapping strategy allows for semantic transformations, where the feature value can be computed using navigation expressions from the contextual type. A feature value transformation is applied in a single direction, either from source to target or from target to source. A programmer can opt to provide feature value transformations in both directions (as in `Task_2` in §2.2), only in one (as in `Task_3` in §2.3), or none at all. The last case corresponds to a simple renaming of feature types as explained above.

For the TTC 2020 case, we are relying on the observation that an EMF-based program can be regarded as a plain Java program. Hence, the given Java program will refer to the source data model and the EMF-based program will refer to the target data model. In the following sections, we discuss the solution to the case in §2, and the evaluation of the solution in §3 using the evaluation criteria proposed in the case, wrapping up with some conclusions in §4.

2. Solution

The solutions to the different tasks of the case are explained below and are available at https://github.com/emf-syncer/ttc20-roundtrip. The solution has been implemented using the language Xtend [4].

2.1. Task 1: create/delete field

In this task, a new feature `age` is added to the class `Person` in the modified data model M_2 and when a `Person` instance is migrated, its feature value is set to `-1`. When the instance is migrated back, this information is lost.

The solution `Task1_M1_M2_M1`, shown in the listing below, illustrates the pattern used in all of the solutions. The EMF-Syncer is used by instantiating the class `EMFSyncer` between a source Java package name list `#['scenario1_v1']` and a target Ecore model in the constructor. All of the solutions use the push-based model in `syncForward`, which is specified by indicating `syncer.syncingStrategy = SyncingStrategy.EAGER`, forcing the migration of the entire source instance.

In the method `migrate`, the statement `syncer.syncForward(person_v1)` migrates the source `Person` instance obtaining the target instance with the `age` feature, which is not initialized until the statement `person_v2.age = -1` is evaluated. In the method `migrateBack`, the statement `syncer.syncBack(person_v2)` migrates the instance back to the source Java program.

The method `modify` is used to implement changes to target model instances that need to be propagated back. This logic is hardcoded in test cases in the benchmark framework. EMF-Syncer needs to track which changes are performed in the target model instance in order to enable incremental back migration. Therefore, this logic has been moved from test cases to task classes.

```
1 class Task_1_M1_M2_M1 extends AbstractTask {
2   val Syncer syncer
3
4   new (EPackage model1, EPackage model2) {
5     super(model1, model2);
6     syncer = new Syncer(#['scenario1_v1'], model2)
7     syncer.syncingStrategy = SyncingStrategy.EAGER
8   }
9
10  override migrate(EObject instance) {
11    val person_v1 = instance as scenario1_v1.Person
12    val person_v2 = syncer.syncForward(person_v1) as scenario1_v2
        .Person
13    person_v2.age = -1
14    return person_v2;
15  }
16
17  override migrateBack(EObject instance) {
18    val person_v2 = instance as scenario1_v2.Person
19    return syncer.syncBack(person_v2) as scenario1_v1.Person
20  }
21
22  override modify(EObject instance) { }
23 }
```

It is important to note that the bidirectional transformation between both data models is completely inferred by the EMF-Syncer automatically in this task. In the back migration, as the `age` feature does not exist in the source program, it is not propagated.

The solution to the symmetric problem `Task1_M2_M1_M2` is achieved by inverting the direction in which the EMF-Syncer is applied to the data models, as seen in the listing below. In this case, the data model `M2` is regarded as the Java program and the data model `M1` as the EMF program. In this case, the value of the `age` feature of a `Person` class is preserved on the back migration because this feature does not exist in `M1` and, therefore, no changes can be applied to the feature `age`. In the following subsections, solutions to symmetric problems are achieved by flipping, as for this task, the direction in which the EMF-Syncer is applied and are not included in the paper.

```
class Task_1_M2_M1_M2 extends AbstractTask {
  val EMFSyncer syncer

  new (EPackage model1, EPackage model2) {
    super(model1, model2);
    syncer = new Syncer(#['scenario1_v2'], model1)
    syncer.syncingStrategy = SyncingStrategy.EAGER
  }

  override migrate(EObject instance) {
    val person_v2 = instance as scenario1_v2.Person
    return syncer.syncForward(person_v2) as scenario1_v1.Person
  }

  override migrateBack(EObject instance) {
    val person_v1 = instance as scenario1_v1.Person
    return syncer.syncBack(person_v1) as scenario1_v2.Person
  }

  override modify(EObject instance) { }
}
```

2.2. Task 2: rename field

In this task, the feature `age` of the class `Person` is renamed to `ybirth`, and its semantics is changed by representing the age and the year of birth, respectively. This renaming involves domain-specific semantics that is not present in the Ecore model, requiring a mapping specification so that the EMF-Syncer can perform the data transformation correctly.

A mapping specification consists of a collection of mappings between feature types by name, `'Person', 'age', /* <-> */ 'Person', 'ybirth'`, and by adding optional feature value transformations as lambda expressions. For example, in the solution, the lambda expression

```
val person_v1 = it as scenario2_v1.Person
Integer.valueOf(Calendar.getInstance().get(Calendar.YEAR) -
    person_v1.age) as Object
```

gets a `Person` instance from the source data model and returns the `ybirth` value. When the EMF-Syncer applies the transformation `syncForward`, the feature value for `Person::ybirth` will be obtained by applying this expression. The rest of the solution follows the same structure as the solution explained in §2.1.

```
class Task_2_M1_M2_M1 extends AbstractTask {
  val EMFSyncer syncer

  // custom mapping strategy: M1 <--> M2
  val public static mapping = new EMFSyncerMapping(
    'Person', 'age', /* <--> */ 'Person', 'ybirth',
    // source to target feature value transformation
    [
      val person_v1 = it as scenario2_v1.Person
      Integer.valueOf(Calendar.getInstance().get(Calendar.YEAR) -
        person_v1.age) as Object
    ],


    // target to source feature value transformation
    [
      val person_v2 = it as scenario2_v2.Person
      Integer.valueOf(Calendar.getInstance().get(Calendar.YEAR) -
        person_v2.ybirth) as Object
    ]
  )

  new (EPackage model1, EPackage model2) {
    super(model1, model2);
    syncer = new EMFSyncer(#['scenario2_v1'], model2,
      newArrayList(mapping))
    syncer.syncingStrategy = SyncingStrategy.EAGER
  }

  override migrate(EObject instance) {
    val person_v1 = instance as Person
    return syncer.syncForward(person_v1) as scenario2_v2.Person
  }

  override migrateBack(EObject instance) {
    val person_v2 = instance as scenario2_v2.Person
    return syncer.syncBack(person_v2) as Person
  }

  override modify(EObject instance) { }
}
```

2.3. Task 3: declare field optional/mandatory

In this task, the multiplicity of a feature type is modified so that the feature is mandatory in one data model and optional in the modified version. This task exposes a difference between a MDE-agnostic program, where it is not possible to know whether a feature is optional in plain Java, and a MDE-aware one, where this information is encoded in the Ecore model. As the EMF-Syncer treats the source program as a plain Java program, it disregards the information contained in the Ecore model, and the logic to transform `null` values in to empty Strings needs to be provided in a custom mapping. The expression `person_v2.name ?: ""` returns an empty String when the `name` of the `Person` instance is null, which is checked using the Elvis operator `?:`. The rest of the data transformation is fully inferred by the EMF-Syncer .

```
class Task_3_M1_M2_M1 extends AbstractTask {
  val EMFSyncer syncer

  // custom mapping strategy: M1 <--> M2
  val public static mapping = new EMFSyncerMapping(
    'Person', 'name', /* <--> */ 'Person', 'name',
    null,
    // target to source feature value transformation
    [
      val person_v2 = it as scenario3_v2.Person
      person_v2.name ?: ""
    ]
  )

  new (EPackage model1, EPackage model2) {
    super(model1, model2);
    syncer = new EMFSyncer(#['scenario3_v1'], model2,
      newArrayList(mapping))
    syncer.syncingStrategy = SyncingStrategy.EAGER
  }

  override migrate(EObject instance) {
    val person_v1 = instance as scenario3_v1.Person
    return syncer.syncForward(person_v1) as Person
```

```
  }

  override migrateBack(EObject instance) {
    val person_v2 = instance as Person
    syncer.syncBack(person_v2) as scenario3_v1.Person
  }

  override modify(EObject instance) {
    val person_v2 = instance as Person
    syncer.track[ person_v2.name = null   ]
    return person_v2
  }
}
```

This solution also contains an example of how to track changes performed in the target instance, in the `modify` method, that need to be migrated back. This change is encoded in the corresponding test case in the original test framework.

2.4. Task 4: multiple edits

This task combines `Task_1` and `Task_2` with the aim of analysing reuse mechanisms that can be employed. The solution below reuses the transformation logic of `Task_1`, as it is handled by the EMF-Syncer automatically, and it reuses the mapping specification of `Task_2`, which is defined as a static field. The rest of the solution is as in §2.1, after renaming the corresponding namespaces.

```
class Task_4_M1_M2_M1 extends AbstractTask {
  val EMFSyncer syncer

  new (EPackage model1, EPackage model2) {
    super(model1, model2);
    syncer = new EMFSyncer(#['scenario4_v1'], model2,
      newArrayList(Task_2_M1_M2_M1.mapping))
    syncer.syncingStrategy = SyncingStrategy.EAGER
  }

  override migrate(EObject instance) {
    val container_v1 = instance as scenario4_v1.Container
    return syncer.syncForward(container_v1) as scenario4_v2.
       Container
  }

  override migrateBack(EObject instance) {
    val container_v2 = instance as scenario4_v2.Container
    syncer.syncBack(container_v2) as scenario4_v1.Container
  }

  override modify(EObject instance) { }
}
```

3. Evaluation

In this section, we provide an evaluation of the solution according to the evaluation criteria proposed in [1].

3.1. Expressiveness

Two test cases are provided for checking the correctness of each task. A test case provides the input for a round-trip migration and the expected output. The test framework has been adapted in order to work with code generated from Ecore models via EMF, which is required by EMF-Syncer . The main properties (namely, `name`, `nsPrefix` and `nsUri`) of the `EPackage` in each Ecore model were updated in order to generate disjoint namespaces. In test cases, loading resources for each Ecore model was modified in order to use the corresponding generated factory for each model instance. Model instances, used as input/output data, were modified to refer to the corresponding `nsUri`. In some cases, the test case also performs a change in the target instance, as in `Task_3`, that needs to be propagated back to the source instance. Such modifications have been encoded in the task itself, in the method `modify`, as the EMF-Syncer needs to track those changes. Such changes refer to implementation details and do not alter the correctness properties checked[2] by the test suite.

Solutions for all of the tasks have been implemented and all of them pass the correctness check.

3.2. Comprehensibility

Task solutions have been implemented using Xtend. However, the EMF-Syncer can be used from Java programs as well. Given that the transformation in most of the solutions is inferred automatically and that solutions use generated code, instead of using the EMF reflection API for accessing/mutating values, we argue that solutions are likely to be more comprehensible than the reference ones.

When custom mappings are required, e.g. in `Task_2` and in `Task_4`, these are defined by instantiating the class `EMFSyncerMapping`, where feature value transformations are defined as Xtend lambda expressions. Such mapping expressions could have been defined similarly in Java as well.

3.3. Bidirectionality

In the solution for most of the tasks, the EMF-Syncer infers both transformations to be applied, `syncForward` and `syncBack`, automatically. In such cases, the programmer does not need to provide a transformation specification and a bidirectional data transformation is being used internally so that Java instances can be migrated to an EMF program, and back again, at run time. A custom mapping specification containing only feature type renamings is fully bidirectional. Feature value transformation expressions are unidirectional though.

Therefore, the EMF-Syncer provides support for bidirectional transformations by default for all of the solutions and accommodates special cases with unidirectional

[2] The test case `task_3_M1_M2_M1_b` had to be updated in order to check that the `name` was migrated back correctly.

	Expressiveness	Comprehensibility	Bidirectionality	Re-usability
Task 1: "Create/Delete Field"				
`Task_1_M1_M2_M1`	2 (2)	2 (2)	1 (1)	n.a.
`Task_1_M2_M1_M2`	2 (2)	2 (2)	1 (1)	n.a.
Task 2: "Create/Delete Field"				
`Task_2_M1_M2_M1`	2 (2)	2 (2)	1 (1)	n.a.
`Task_2_M2_M1_M2`	2 (2)	2 (2)	1 (1)	n.a.
Task 3: "Create/Delete Field"				
`Task_3_M1_M2_M1`	2 (2)	2 (2)	1 (1)	n.a.
`Task_3_M2_M1_M2`	2 (2)	2 (2)	1 (1)	n.a.
Task 4: "Create/Delete Field"				
`Task_4_M1_M2_M1`	2 (2)	2 (2)	1 (1)	4 (4)
`Task_4_M2_M1_M2`	2 (2)	2 (2)	1 (1)	4 (4)
	$\sum$: 16 (16)	$\sum$: 16 (16)	$\sum$: 8 (8)	$\sum$: 8 (8)

Table 1
Summary of evaluation results.

feature value transformations, which correspond to a fraction of the data migration to be performed.

3.4. Re-usability

Re-usability is internalized in the EMF-Syncer by inferring transformations automatically. That is, when a user does not need to provide a mapping specification, the transformation logic in the EMF-Syncer is reused for any data model change. For example, reuse of the logic transformation in `Task_1` falls under this category.

On the other hand, as EMF-Syncer is a JVM library, a programmer can rely on reuse mechanisms provided by the JVM language of choice. For example, reuse of the transformation logic in `Task_2` falls under this category. The mapping specification defined for `Task_2` is reused by calling a static field and by using a common interface for the class `Person`, which has been implemented using inheritance in `Task_4`.

3.5. Performance

We have run the performance tests both for the reference solution and for the EMF-Syncer solution on a MacBookPro11,5 Core i7 2.5 GHz, with four cores and 16 GB of RAM. The runtime results (in ms.) obtained are displayed in Fig. 1. The EMF-Syncer solution exhibits a linear growth with respect to the number of iterations, as instructed in the case benchmark. However, the EMF-Syncer solution is more efficient than the reference one thanks to its support for incremental migration of changes. While the reference solution takes about 31 s. for 2 million iterations, the EMF-Syncer solution takes about 9 s, with an improvement factor of 70%. The incremental propagation relies on a traceability model that caches relevant data at run time. An analysis of memory consumption is left for future work as it was not part of the case benchmark.

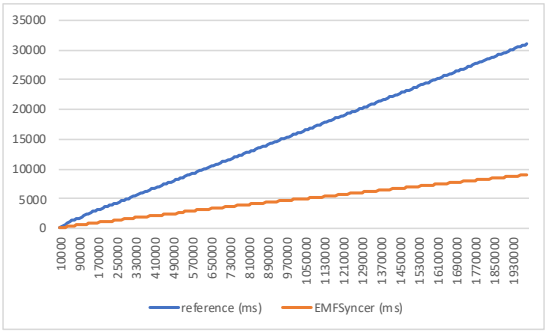

Figure 1: Performance results (ms.) along iterations

4. Conclusions

Following from the justification of the evaluation criteria in the section above, the results are summarized in Table 1, where the score X out of Y is expressed as $X(Y)$. Overall, this solution shows that the EMF-Syncer helps in achieving a competitive trade-off between data migration specification and performance. On the one hand, the automatic inference of data transformations between different object-oriented data models reduces the need for specifying data transformations. When these have to be specified, a programmer can rely on their programming skills, using a JVM programming language, for reusing transformation logic. On the other hand, EMF-Syncer can be used as an efficient data migration service at run time. For example, the EMF-Syncer solution is faster than the reference solution developed in raw Java. EMF-Syncer can be used for building more scalable solutions, involving very large models, by using the pull-based model for propagating changes in `syncForward` only when they are required in the target program.

References

[1] L. Beurer-Kellner, J. von Pilgrim, T. Kehrer, Round-Trip Migration of Object-Oriented Data Model Instances, in: Proceedings of the 13th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences, CEUR Workshop Proceedings, CEUR-WS.org, 2020.

[2] A. Boronat, Code-first model-driven engineering: On the agile adoption of mde tooling, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019), San Diego, CA, November 11-15, ACM, 2019.

[3] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework 2.0, 2nd ed., Addison-Wesley Professional, 2009.

[4] T. E. Foundation, Xtend (official web page), 2018. http://www.eclipse.org/xtend/.