Generating Database Migrations for Evolving Web Applications

Sander D. Vermolen Guido Wachsmuth Eelco Visser
Delft University of Technology, The Netherlands
{s.d.vermolen, g.h.wachsmuth, e.visser}@tudelft.nl

Abstract

WebDSL is a domain-specific language for the implementation of dynamic web applications with a rich data model. It provides developers with object-oriented data modeling concepts but abstracts over implementation details for persisting application data in relational databases. When the underlying data model of an application evolves, persisted application data has to be migrated. While implementing migration at the database level breaks the abstractions provided by WebDSL, an implementation at the data model level requires to intermingle migration with application code. In this paper, we present a domain-specific language for the coupled evolution of data models and application data. It allows to specify data model evolution as a separate concern at the data model level and can be compiled to migration code at the database level. Its linguistic integration with WebDSL enables static checks for evolution validity and correctness.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; H.2.1 [*Database Management*]: Logical Design

General Terms Languages

Keywords Evolution, Domain Specific Language, Data Migration, Web Application

1. Introduction

WebDSL is a domain-specific language for the implementation of dynamic web applications with a rich data model [16]. It provides developers with object-oriented data modeling concepts. These concepts abstract over implementation details for persistence. These details are added in a two-step compilation process. In the first step, the WebDSL compiler generates application code in an object-oriented general purpose programming language, which is Java. To achieve persistence, the generated code relies on the Hibernate framework. This framework realizes an object-relational mapping (ORM): Application data is kept in objects at runtime but is persisted in a relational database. In the second step, the generated application code is compiled and the persistence framework generates a relational database schema. When deploying the application, a relational database management system (RDBMS) generates an initial, empty database from this schema. The deployed application will interact with the RDMBS to store and to retrieve its data.

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.

Problem Statement. As any other software, web applications and their data models evolve. An evolved application has to be recompiled and redeployed. During recompilation, the persistence framework generates a new database schema. Typically, the original database no longer complies with the new schema and original application data cannot be accessed from the evolved application anymore. During redeployment, the RDBMS instead generates a new initial database from the new schema. But original application data is a valuable asset. It needs to be migrated to co-evolve with the application and its data model.

Implementing migrations at the database level breaks the abstractions provided by WebDSL. Developers have to be aware of the persistence framework and its ORM to make sure that the migrated database complies with the new schema. They also have to be aware of the RDBMS to provide details such as character set definitions, collations, and storage engines.

To avoid breaking abstractions, migrations can be implemented at the data model level in WebDSL. Since the generated code will make extensive usage of the ORM, migration does not scale to large amounts of data and is typically performed lazily. The application migrates original data only when it needs to access this data. As a consequence, the original data model has to remain part of the evolving data model and application code is intermingled with migration code. Maintenance of data model, application code, and migration code becomes harder with every new evolution step.

Contribution. In our previous work, we compiled an extensive catalog of coupled operators for the evolution of object-oriented data models [8]. These operators couple common evolution steps at the data model level with their corresponding migrations at the data level. In this paper, we focus on the implementation of these operators in Acoda, a tool for the coupled evolution of WebDSL data models and databases.

Acoda provides a domain-specific language for specifying data model evolution as a separate concern at the data model level. Its IDE offers static checks for evolution validity and correctness. While evolution validity ensures that an evolution can be applied to the original data model, evolution correctness secures that the evolution yields the evolved data model.

Acoda implements coupled operators as a mapping from evolution steps to migration code in SQL. In this paper, we discuss this mapping for particular operators in detail, including complex operators that work along the inheritance hierarchy or over references. Thereby, we distinguish three kinds of migrations. First, *schema modifications* change only the database schema. Second, *conservative migrations* rearrange data without data loss. Third, *lossy migration* supports potential data loss on purpose.

Outline. We briefly introduce WebDSL's data modeling concepts and its ORM in the next section. In Section 3, we discuss evolution specification. In Sections 4 to 6, we address the generation of migration code for selected operators in detail. We conclude the paper with a discussion in Section 7.

2. WebDSL

WebDSL is a domain-specific language for the development of dynamic web applications that integrates data models, user interface models, actions, validation, access control, and workflow [16]. The WebDSL compiler verifies the consistency of web applications and generates complete implementations in Java. In this section, we focus on WebDSL's data modeling concepts and the ORM underlying the generated Java code.

Data modeling. A data model definition in WebDSL features *entity* declarations, which comprise a name and a set of properties. An entity declaration might inherit from another entity declaration, indicated with `:` . Each *property* has a name and a type. We distinguish two kinds of properties: Value properties, indicated with `::` , and associations, indicated with `→` . For value properties, WebDSL supports basic data types such as `Bool` and `String`, but also domain-specific types such as `Email`, `Secret`, and `WikiText`, which all provide additional functionality. Associations refer either to entities declared in the data model (*single-valued*) or to a **`Set`** or **`List`** thereof (*multi-valued*).

Figure 1 (top) shows a data model for a publication management application similar to Researchr[1]. It models publications written by authors and a special type of publication, namely the published volume. Additionally, users can register and create personal bibliographies, which are collections of publications.

Object-relational Mapping. WebDSL's data modeling concepts abstract over implementation details for persistence. These details are added by the WebDSL compiler which addresses Hibernate as a persistence framework. At runtime, application data is kept in objects which are stored persistently in a relational database. There is a database table for each hierarchy of entities, named after the root entity declaration in a hierarchy. Throughout the paper, we will call these tables *hierarchy tables*. In the running example, there will be four tables `_User`, `_Bibliography`, `_Publication`, and `_Author`. Each of these tables has at least two columns: `id` stores object ids and acts as the primary key of the table while `DISCRIMINATOR` is used to distinguish which entity in the hierarchy is instantiated by an object. Object ids are implemented by universally unique identifiers (UUIDs) and are therefore database-wide (and beyond) unique.

Additional columns are added for each value property and for each single-valued association declared in one of the entities in an entity hierarchy. Since columns for single-valued associations will store the ID of a referred object, they act as implicit references. The implicit references are made explicit by a foreign key to the `id` column in the table corresponding to the type of an association. The RDBMS enforces foreign keys by preventing (or canceling) database operations that break integrity. The `_Bibliography` table will have three columns: `id` (primary key), `DISCRIMINATOR`, and `Bibliography_owner` (foreign key to `id` in `_User`).

Multi-valued associations are stored in separate *connection tables*. The names of these tables are composed from the names of the declaring entity, the association, and the association type. Each connection table has two columns to store pairs of object ids (referring and referred object). Both columns are foreign keys to the `id` column of the table corresponding to the declaring entity respectively the association type. A multi-valued association can either be a **`Set`** or a **`List`** . For sets, we place a primary key on the two columns, since we may not store a pair of objects twice. For lists, an additional index column is needed to persist order. Here we place a primary key on the combination of declaring entity reference and the index, since there can just be one reference per position in a list. For example, the association `Publication.authors` is stored in a connection table `Publication_authors_Author` with three columns `_Publication_id` (foreign key to `id` in `_Publication`), `Publicationauthorindex`, and `authors_id` (foreign key to `id` in `_Author`), where the first two columns act as the primary key.

[1] Researchr is a web application for finding, collecting, sharing, and reviewing scientific publications: `http://researchr.org`.

original data model

```
entity Author {
  name :: String
}

entity User {
  email    :: Email
  password :: Secret
  public   :: Bool
}

entity Bibliography {
  owner        → User (not null)
  publications → Set<Publication>
}

entity Publication {
  key      :: String
  title    :: String
  abstract :: WikiText
  authors  → List<Author>
}

entity PublishedVolume : Publication {
  publisher :: String
}
```

evolved data model

```
entity Person {
  alias → Set<Alias> (not empty)
  email :: Email
}

entity Alias {
  name :: String (id)
}

entity User : Person {
  password :: Secret
}

entity Bibliography {
  public       :: Bool
  owner        → User (not null)
  publications → Set<Publication>
}

entity Publication {
  registrant → User
  key        :: String
  title      :: String
  abstract   :: WikiText
  authors    → List<Person>
}

entity PublishedVolume : Publication {
  editors   → List<Person>
  publisher :: String
}
```

evolution model

```
1 create Publication.registrant → User;
2 collect Bibliography.public over owner;

3 rename entity Author to Person;
4 create PublishedVolume.editors → List<Person>;
5 add super Person to User;
6 pull up Person.email;

7 extract entity Alias{name::String} from Person as alias;
8 make Alias.name id;
9 generalize Person.alias to Set;
```

Figure 1. Running example

3. Modeling Data Model Evolution

Typically, the evolution of a data model is only implicitly defined by its original and evolved version. For example, the middle part of Figure 1 shows an evolved version of the data model from the top of the figure. In this section, we discuss means to model this evolution explicitly.

Coupled Operators. Informally, the example evolution follows three stages: First, bibliography management is extended by allowing users to submit new publications, hence they are linked to publications as `registrant` and can now individually set bibliography visibility. This requires adding an association from `User` to `Publication` and a `public` property to bibliographies. The latter is collected from the `owner` of a bibliography as not to lose the user settings. Second, the system is refactored to support editors. This requires addition of an `editors` association, as well as renaming `Author` to a more general `Person`. Consequently, `User` can become a sub entity of `Person`, since they may also be editor or author of publications. The `email` of users is then generalized to be able to store email addresses for editors and authors. Third, the system is extended to support people (authors, editors, or users) to have different name aliases. Therefore, a person's `name` is extracted into a new entity, in which names are stored uniquely.

We can model this evolution as a sequence of coupled operator applications [8, 18]. At the data model level, coupled operators capture common evolution steps. Thereby, they go beyond simple creations, changes, and deletions of entities and properties. For example, the evolution model from the bottom part of Figure 1 includes the collection of a property over an association, the pull-up of a property into a parent entity, and the extraction of an entity. Each of these operators couples the evolution step at the data model level with a corresponding migration. This allows us to compile evolution models into migration code for the database level.

Linguistic Integration. The language for evolution models is linguistically integrated with WebDSL. It reuses WebDSL's data modeling concepts and parts of their syntax definition. For example, constructs for property and entity creation reuse the syntax for properties and entities. An evolution model includes references to the original and evolved data model. Static checks ensure evolution validity and correctness with respect to these data models. For evolution validity, preconditions for operator applications are checked in the context of the original data model [15]. These preconditions secure that the evolution can be applied to the original data model. For evolution correctness, it is checked whether the evolution maps the original data model to the evolved data model.

Migration. To migrate the original database, each operator application in an evolution model is compiled to its corresponding migration code. Thereby, the compiler follows the same ORM as the WebDSL compiler, namely Hibernate. This ensures that migrating the original database and generating an initial schema for the evolved data model will result in the same database schema. Furthermore, the compiler is aware of the RDMBS and generates details such as character set definitions, collations, and storage engines[2].

In the following sections, we discuss database migration for selected coupled operators. Thereby, we distinguish three kinds of migrations. Operators such as property and entity creation only require *schema modification*. Their corresponding migrations affect the database schema but not the stored data. We discuss such operators in Section 4. Many other operators such as entity renaming, entity extraction, super addition, or cardinality generalization allow for *conservative data migration*. Their corresponding migrations affect both the database schema and the stored data. But the stored data is completely preserved during migration, no data is lost. We discuss such operators in Section 5. Only few operators such as property collection or property identification require *lossy migration*. Their corresponding migrations may not preserve the stored data completely. Some data may be lost intentionally during migration. We discuss such operators in Section 6.

[2] In the examples, we omit these details for readability.

4. Schema Modification

Schema modifying migrations change the database schema, but leave the persistent data untouched. They generally allow for more information to be stored and are thereby most commonly needed while extending application functionality.

4.1 Property Creation

In earlier work [8], we identified two coupled operators for property creation: one for value properties and one for associations. But in WebDSL and its Hibernate configuration, single-valued associations and multi-valued associations are dealt with differently. The first is stored inside the containing entity's table, the second is stored in its own connection table. Thus, Acoda provides three different coupled operators for property creation: one for value properties, one for single-valued associations, and one for multi-valued associations.

When we create a new value property in a data model, the original database schema is missing a column for this property. To be precise, the hierarchy table corresponding to the entity containing the new property is missing a column. We need to create the missing column in order to migrate the original database.

For the creation of a new single-valued association, the migration is similar. Again, the hierarchy table corresponding to the containing entity is missing a column for storing ids of the associated entity. Additionally, a foreign key needs to be created to enforce validity. This foreign key needs to point to the `id` column of the hierarchy table corresponding to the associated entity.

Example. Acoda generates the following migration for the creation of the `registrant` association in the running example:

```
1 create Publication.registrant → User;
```

```
ALTER TABLE _Publication
   ADD COLUMN `Publication_registrant`
      VARCHAR(32) default NULL,
   ADD CONSTRAINT f_Publication_registrant
      FOREIGN KEY `f_Publication_registrant`
         (Publication_registrant)
      REFERENCES _User (id);
```

It consists of a single SQL statement altering the `_Publication` table. First, it adds a new column `Publication_registrant` to store the association. Afterwards, it constrains this column with a foreign key to `id` in `_User`.

In contrast to single-valued references, multi-valued references are stored in separate connection tables. When we create a new multi-valued association in a data model, the original database schema is missing a table for this association. We need to create this table in order to migrate the original database.

Example. Acoda generates the following migration for the creation of the `editors` association:

```
4 create PublishedVolume.editors → List<Person>;
```

```
CREATE TABLE `PublishedVolume_editors_Person` (
   `_PublishedVolume_id` VARCHAR(32) default NULL,
   `_editors_id` VARCHAR(32) default NULL,
   `PublishedVolumeeditorsindex` integer,
   PRIMARY KEY (`_PublishedVolume_id`,
      `PublishedVolumeeditorsindex`),
   INDEX `forward_lookup`
```

```
      (_PublishedVolume_id(14)),
   CONSTRAINT `f_PublishedVolume_editors_b`
      FOREIGN KEY `f_PublishedVolume_editors_b`
      (_PublishedVolume_id)
      REFERENCES _Publication (id),
   CONSTRAINT `f_PublishedVolume_editors_f`
      FOREIGN KEY `f_PublishedVolume_editors_f`
      (_editors_id)
      REFERENCES _Person (id)
);
```

It comprises a single statement creating a table connecting `_PublishedVolume` to `_Person`. The table has three columns to store ids of published volumes, ids of persons, and list indices since order does matter. For a published volume and an index, the associated editor needs to be unique. Thus, the published volume and the index form the primary key of the table. Validity of the two columns which store ids is ensured by foreign keys. These point to the `id` columns in the connected tables. In order to support efficient use of the connection table, database indices are generated for the primary key, allowing efficient single editor lookup, and for the published volumes column, allowing efficient collection of the complete list of editors for a published volume.

4.2 Entity Creation

Similar to property creation, Acoda provides different coupled operators for the creation of a new entity: one for entities that do not extend another entity and one for entities that do.

When we create a new entity which does not extend another entity, the original database is missing a hierarchy table for this entity and connection tables for its multi-valued associations. We need to create these tables in order to migrate the original database. Since we explained the creation of connection tables already in the previous section, we only focus on the hierarchy table. Following Hibernate, this table needs to be named like the entity and needs to provide two columns `id` and `DISCRIMINATOR` as well as additional columns for value properties and single-valued associations. Columns for single-valued associations need to be constrained by foreign keys.

When we create a new entity which extends another entity, the original database is missing columns for its value properties and single-valued associations and connection tables for its multi-valued associations. The columns are missing in the hierarchy table of the extended entity. Thus, the migration is the same as for creating the properties of the new entity. Creating its value properties and single-valued associations will add the missing columns while creating its multi-valued associations will add the missing connection tables.

Figure 2 presents creation of an **`entity`** `A:B` with single- and multi-valued features `sf 1 .. sf n` and `mf 1 .. mf m` graphically. The figure above the dashed line shows the database before migration, the figure below the dashed line after migration. Each array denotes a table, each cell within an array a column. An open cell denotes columns which were already present before modification and remain unaltered. A solid double arrow denotes a uniqueness key, a dashed double arrow denotes a database index, and a single arrow denotes a foreign key. `A*` denotes the root entity in the hierarchy of entity `A`, `t(a)` denotes the target type of an association `a`. The id columns are always unique. We therefore omit the double solid uniqueness arrow on id columns.

5. Conservative Data Migration

Conservative migrations are needed when the domain of an application shifts or expands. They change the schema and rearrange data but do not lose information. Conservative migrations are most common in practice, yet tedious and error-prone to write manually.

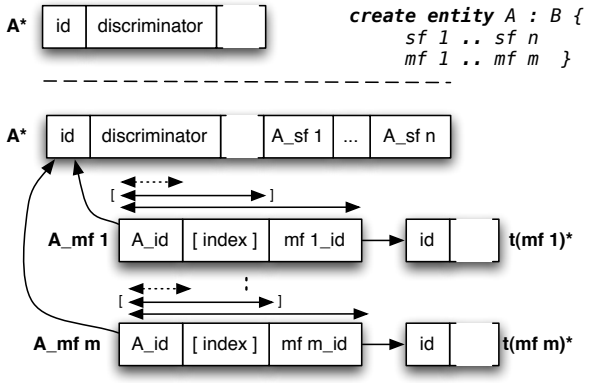

Figure 2. Database modification for entity creation.

5.1 Entity Renaming

In schema generation, entity names influence table and column names in hierarchy and connection tables as well as associated foreign keys. When an entity is renamed, these names need to be updated and foreign keys need to be recreated. More specifically, renaming entity `A` requires the following schema modifications:

1. Drop foreign keys for single- and multi-valued associations in `A`
2. Drop foreign keys for single- and multi-valued associations of type `A`
3. Rename the hierarchy table for `A` if `A=A*`
4. Rename columns for value properties and single-valued associations in `A`
5. Rename connection tables for multi-valued associations in `A`
6. Rename columns for multi-valued associations in `A`
7. Rename columns for multi-valued associations of type `A`
8. Create foreign keys for single- and multi-valued associations in `A`
9. Create foreign keys for single- and multi-valued associations of type `A`

However, entity names are also used as discriminator, distinguishing between different entities in a hierarchy. An entity rename therefore needs to migrate the data inside the hierarchy table for `A`, by replacing the old entity name in the `DISCRIMINATOR` column by the new entity name.

Example. Acoda generates the following migration for renaming `Author` to `Person`:

```
3 rename entity Author to Person;
```

```
ALTER TABLE Publication_authors_Author
   DROP FOREIGN KEY `f_Publication_authors_f`;
ALTER TABLE _Author
   RENAME _Person;
ALTER TABLE Publication_authors_Author
   RENAME Publication_authors_Person,
   ADD CONSTRAINT `f_Publication_authors_f`
      FOREIGN KEY `f_Publication_authors_f` (_authors_id)
      REFERENCES _Person (id),
UPDATE _Person
   SET DISCRIMINATOR = "Person"
   WHERE DISCRIMINATOR = "Author";
```

First, the foreign key for the `Publication.authors` association is dropped, after which the hierarchy table `_Author` can be renamed. Next, the connection table for the association is renamed and the dropped foreign key is recreated. Finally, the object discriminators are updated.

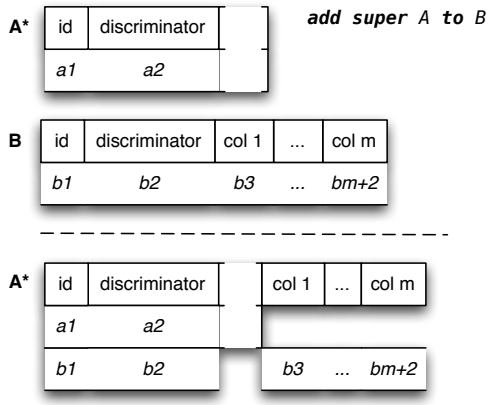

Figure 3. Database modification for super type addition.

5.2 Super Addition

When the original application models two inheritance-unrelated entities, separate tables are used to store the inheritance trees of both. When the application evolves by adding a super entity joining the two inheritance trees, the target application only uses a single table to store both entities. To support the new application, the original tables and associated data needs to be merged. The schema modifications are presented graphically in Figure 3. The migration of adding super entity `A` to entity `B` comprises the following steps:

1. Expand the table for `A*` by all single-valued properties in the inheritance tree of `B` (`inh(B)`)
2. Create foreign keys for outward single-valued associations in `inh(B)`
3. Copy single-valued data from the old table for `B` to the table for `A*`
4. Drop foreign keys for outward (multi-valued and single-valued) associations in `inh(B)`
5. Create foreign keys for outward multi-valued associations in `inh(B)`
6. Drop foreign keys for inward associations to `inh(B)`
7. Create foreign keys for inward associations to `inh(B)`
8. Drop the old table for `B`

Step 1 creates the new space to store data for `B` and its sub entities inside the table, which was originally only used for `A*` and its sub entities. Step 2 creates foreign keys pointing away from the table of `A*`. Step 3 prevents any loss of data. Steps 5 and 7 create foreign keys pointing to the table of `A*`, which work on the copied data. Steps 4 and 6 drop all foreign keys, that point to the table for `B`, to prevent breaking the database integrity. Finally, step 8 deletes the old data. The order of steps is crucial, it targets to maximize the number of constraints at any point in migration: Foreign keys for outward single-valued associations are added before copying data, since they point away from the table for `A*` and thereby also hold on an incomplete (or empty) set of `B` records. Foreign keys for outward multi-valued associations and inward associations are created after copying, since they point to the table for `A*` and therefore require a complete data set. The foreign keys are dropped before the data is dropped to prevent them from breaking and the foreign keys are dropped before they are recreated to prevent name clashes. Note that except for their foreign keys, any connection table associated to `inh(B)` remains unaltered.

Example. In the running example, `Person` becomes super entity of `User`. Following the scheme outlined above: `User` has two single-valued properties `email` and `password`, which are added to the `Person` table in step 1. Both these properties are attributes, hence step 2 can be omitted. Next, the user data is copied from the `User` table to the `Person` table in step 3. Step 4 can again be omitted. User has one inward association `registrant` from `Publication`, whose new foreign key is added in step 5 and whose old foreign key is dropped in step 7. Step 6 can again be omitted and step 8 drops the old user data. The steps are formalized in the following migration:

```
5 add super Person to User;
```

```
ALTER TABLE _Person
   ADD COLUMN `User_email` VARCHAR(255) DEFAULT '';
   ADD COLUMN `User_password` VARCHAR(255) DEFAULT '';
INSERT INTO _Person
   (id,DISCRIMINATOR, version_opt_lock,
      User_email, User_password)
   SELECT id,DISCRIMINATOR,
      version_opt_lock,_email,_password
   FROM _User;
ALTER TABLE _Publication
   DROP FOREIGN KEY `f_Publication_registrant`;
ALTER TABLE _Publication
   ADD CONSTRAINT `f_Publication_registrant`
   FOREIGN KEY `f_Publication_registrant`
   (Publication_registrant)
   REFERENCES _Person (id);
DROP TABLE _User;
```

5.3 Entity Extraction

To enrich a data model, an entity may need to be extracted from another entity. During entity extraction, a new entity is created using some or all of the properties of an existing source entity. A single-valued association is created to link objects of the two entities. An example entity extraction can be found in the running example, where `Alias` is extracted from `Person`, using a new association `alias`. We distinguish the following steps in a migration for extracting entity `B` from `A` as `a`:

1. Adapt the schema to store `B`
2. Add a column for `a` to the table for `A*`
3. Generate new identifiers in the column for `a`
4. Copy `a` as id and other single-valued columns in `B` from the table for `A*`
5. Create a foreign key for `a`
6. Drop the old columns in the table for `A*`
7. Move the data for multi-valued properties in `B` and update their ids using the mapping provided by `a`

Step 1 comprises a migration for creating an entity, as discussed in Section 4.2. Step 2 adds a column, but leaves out its foreign key. Step 3 generates new ids, which can be sequentially numbered, or as in our case UUIDs. Step 4 then performs the extraction for all single-valued data, by copying their columns including the newly generated ids and a discriminator (`'B'`) to the new table. As the identifier duplication now validates the foreign key, it can be created in step 5. Step 6 then drops the old single-valued data from the table for `A*`. Finally, step 7 moves the multi-valued data to new connection tables, which were created in step 1. This data references `A` objects, whereas they should now be referencing `B` objects, therefore there links need to be updated using the mapping specified in the table for `A* (id, A_a)`. After moving the multi-valued data, the old connection tables are dropped. Note that step 4 moves each property across association `a` to `B`. We could therefore have reused the migration generation for moving properties, yet this would yield an inefficient migration. Copying all data in one pass over the table for `A*` is more efficient then separate passes for each of the single-valued properties in `B`.

Figure 4 shows the database before and after migration. The data set identifiers are generated (step 3 above) and the data set for

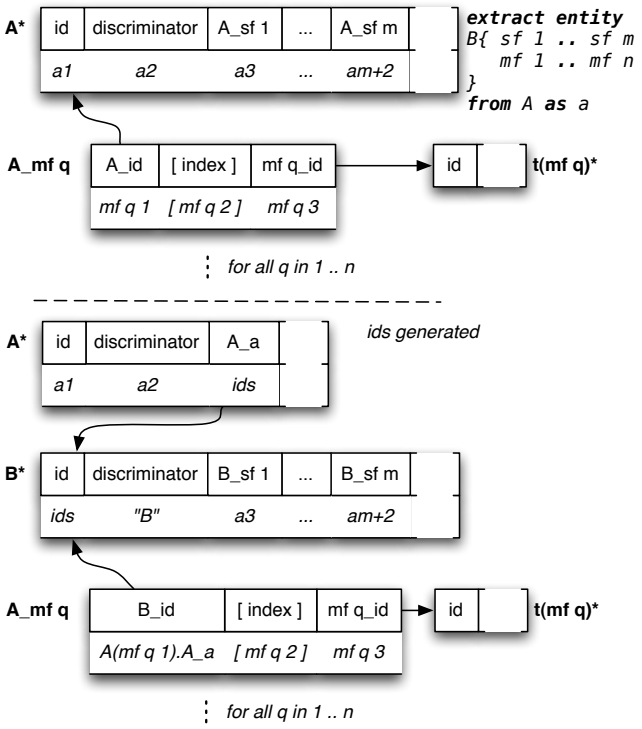

Figure 4. Database modification for entity extraction.

B_id is obtained by applying the mapping from A objects to B objects (step 7).

Example. In the running example, we extract entity Alias and its name property from Person , while creating an association alias. To adapt the database, we generate the migration shown below. Step 7 above is not represented, since Alias has no multi-valued properties.

```
7 extract entity Alias{name::String} from Person as alias;
```

```
CREATE TABLE IF NOT EXISTS `_Alias` (
    `DISCRIMINATOR` VARCHAR(255) default '',
    `id` VARCHAR(32) default NULL,
    `Alias_name` VARCHAR(255) default '',
    PRIMARY KEY (`id`)
);
ALTER TABLE _Person
    ADD COLUMN `Person_alias` VARCHAR(32)
        default NULL;
UPDATE _Person
    SET Person_alias = UUID();
INSERT INTO _Alias
    SELECT "Alias", Person_alias, Person_name
    FROM _Person;
ALTER TABLE _Person
    ADD CONSTRAINT `f_Person_alias`
    FOREIGN KEY `f_Person_alias` (Person_alias)
    REFERENCES _Alias (id);
ALTER TABLE _Person
    DROP COLUMN Person_name;
```

5.4 Maximum Cardinality Generalization

During the lifetime of an application, attributes often get generalized to expand the application's functionality. One type of property generalization is increasing its maximum cardinality. Any multi-valued cardinality uses the same database structure, its exact number is irrelevant. However, a single-valued association is represented as a column, whereas a multi-valued association as a connection table. In the running example, a person's alias is stored within the Person table before step 7 and stored in a connection table afterwards. To support such generalization, we need to generate a migration, which first creates the connection table, then moves the data from the main table to the connection table and subsequently drops the old column.

Example. For generalizing the maximum alias cardinality in the running example, we generate the migration below. The first statement creates a connection table as discussed in Section 4.1. The second statement inserts the old data into the new connection table. The last two statements drop the old column by first dropping the foreign key and then dropping the column itself.

```
9 generalize Person.alias to Set;
```

```
CREATE TABLE IF NOT EXISTS `Person_alias_Alias` (
    `_Person_id` VARCHAR(32)default NULL,
    `_alias_id` VARCHAR(32)default NULL,
    INDEX `forward_lookup` (_Person_id(14)),
    CONSTRAINT `f_Person_alias_b`
        FOREIGN KEY `f_Person_alias_b` (_Person_id)
        REFERENCES _Person (id),
    CONSTRAINT `f_Person_alias_f`
        FOREIGN KEY `f_Person_alias_f` (_alias_id)
        REFERENCES _Alias (id)
);
INSERT INTO Person_alias_Alias
    SELECT id, Person_alias
    FROM _Person
    WHERE Person_alias IS NOT NULL;
ALTER TABLE _Person
    DROP FOREIGN KEY f_Person_alias;
ALTER TABLE _Person
    DROP Person_alias;
```

5.5 Property Pull-Up

For pulling up a property, Acoda provides different migrations for value properties, single-valued associations, and multi-valued associations. Value properties as well as single-valued associations are stored inside the inheritance hierarchy table. A property is pulled up from each of the sibling entities inside the hierarchy. The pulled up property is stored in one database column. During migration, the values for the different sibling columns need to be combined. This is achieved by creating the new column A_f, copying the data sets of each of the sibling properties separately and dropping the sibling properties afterwards. Figure 5 presents single-valued pull up. The pulled up data (a12 to an2) is merged to form a new column A_f. When associations are pulled up, the old foreign keys are dropped (arrows in figure) and a single foreign key is created along with the new column A_f.

Example. In the running example, email is pulled up from User to Person . Email is a single-valued property and User has no sibling entities. Thus, for the example, we need to merge a single column with no foreign key, which amounts to creating a new column, copying the data and dropping the old column:

```
6 pull up Person.email;
```

```
ALTER TABLE _Person
    ADD COLUMN `Person_email`
    VARCHAR(255) DEFAULT '';
UPDATE _Person
    SET Person_email = User_email
    WHERE DISCRIMINATOR='User';
ALTER TABLE _Person
    DROP COLUMN `User_email`;
```

When pulling up a multi-valued association, the set of sibling associations is stored in a collection of connection tables. These need to be merged into a new table which has column names and foreign keys adapted to the new containing type. In contrast to single-valued associations, merging of multi-valued associations comprises a union of the sibling data sets and can thus be done in one SQL statement.

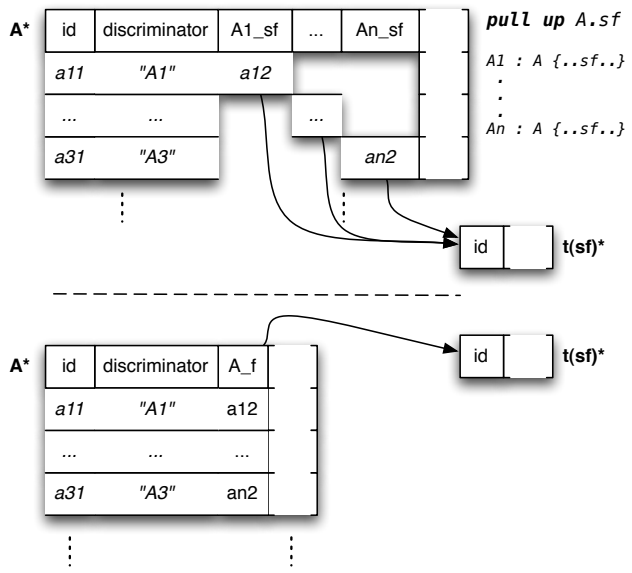

Figure 5. Database modification for single-valued property pull up.

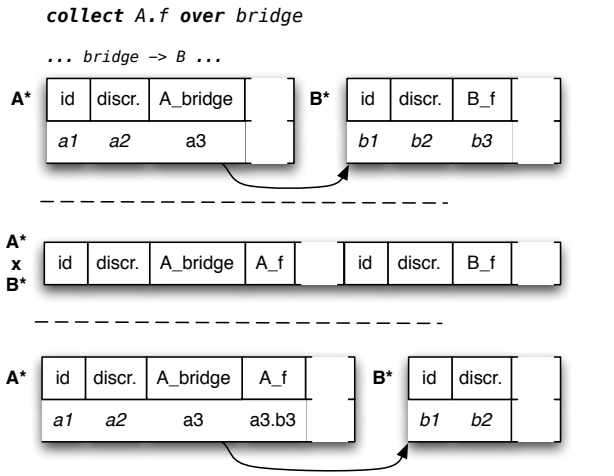

Figure 6. Database modification for property collection.

6. Lossy Migration

Although data loss is generally not desirable, when correcting design flaws it can often not be avoided. Additionally, in many cases a migration may in theory potentially cause data loss, yet in practice for many databases this will not actually be the case.

6.1 Property Collection

It is common for properties to be repositioned during the life-span of an application. They can be repositioned across an inheritance relation (e.g. **`pull up`**), but can also be repositioned across an association. In WebDSL, associations are directed. When repositioning a property in the direction of the association, we speak of *moving* a property, when repositioning opposite to the association direction, we speak of *collecting* a property. When a property is repositioned across an association, we call the association a *bridge*.

There are two main reasons for collecting properties: First, the application may use numerous dereferences to access a property, in which case the dereference can be made permanent by collecting the property. Second, the property might no longer logically belong to the referred object but to the referring object. This is the case in our running example: We want to distinguish for each bibliography if it is public or not. In the original data model, the distinction is made only on a per user basis. Thus, the corresponding property `public` needs to be collected from `User` to `Bibliography`, using `owner` as a bridge.

Property collection may cause loss of data, since the bridge may not be surjective. There may be users who have set their public field but do not have a bibliography. To adapt a database to a collected single-valued property `f` in `A` from `B` across single-valued association `bridge`[3], we first create the new column to store `f`. Next, we join the tables for `A*` and `B*` (we compute their cross product) and filter the result on records where the bridge holds `(A.bridge = B.id)` . Then we copy the old column for `f` to the new column for `f` in the cross product result. Finally we drop the old column for `f`. If `f` is an association, its new foreign key needs to be created along with creating its new column and its old foreign key needs to be dropped before dropping the old column.

[3] Note that `A` and `B` could be the same type

Note that the database index on the (primary) id column of the table for `B` ensures that the table join can be computed efficiently.

Figure 6 shows the process graphically, in which the middle stage represents the intermediate signature during update. During migration, data is typically duplicated: A user can have multiple bibliographies, each of which gets the same public value.

Example. To collect `public` from `User` to `Bibliography` in our running example, we generate the migration below. The first and last statement create and delete columns to store the `public` property. The second statement copies (and duplicates) the information by updating the `Bibliography` and `User` table joined together using `owner` as criterium.

```
collect Bibliography.public over owner;
```

```
ALTER TABLE _Bibliography
   ADD COLUMN `Bibliography_public` BIT(1)
   DEFAULT FALSE;
UPDATE _Bibliography target, _User source
   SET target.Bibliography_public = source.User_public
   WHERE target.Bibliography_owner = source.id;
ALTER TABLE _User
   DROP User_public;
```

There are different migrations for collecting multi-valued properties and for collecting properties across a multi-valued bridge. In both cases, the migration is extended by including connection tables. When collecting a property across a multi-valued bridge, we need to extend the join above by the connection table representing the bridge. When collecting a multi-valued property, the property is represented as a connection table and we can thereby make a new connection table by rewriting the connection table's reference to `B` into a reference to `A`, using the bridge. To apply the rewriting efficiently, a database index on the bridge needs to be generated first.

6.2 Property Identification

When a property is kept unique by the application, yet is not modeled as such, it can be made unique to ensure correctness of the application logic. Also, when data is stored redundantly, it can be compacted by enforcing uniqueness of redundant properties. The latter is the case in the running example, where multiple aliases with the same name exist after entity extraction. By making an alias' name unique, only one object would be needed per name.

Although the schema generated for the new application would match the original schema, the application logic assumes property uniqueness, whereas this is not guaranteed by the database. The original database may contain duplicate values. Migration needs

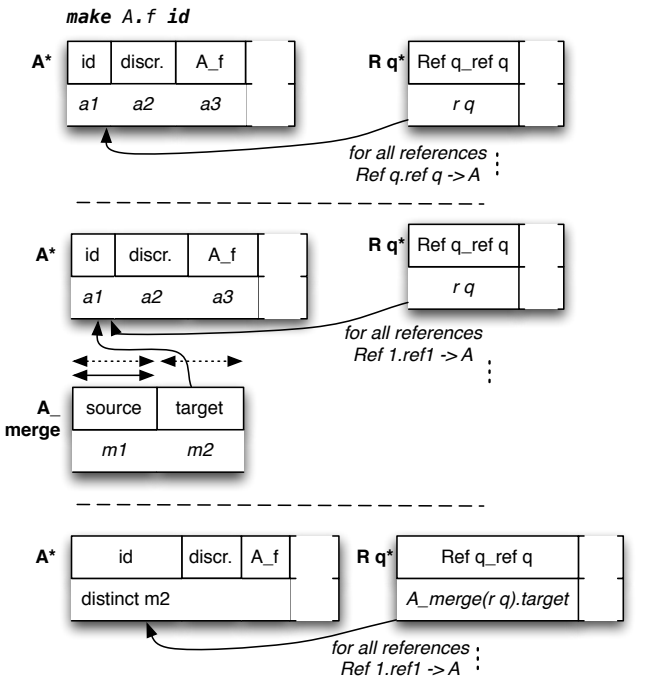

Figure 7. Database modification for attribute identification.

to resolve these duplicates as to ensure uniqueness. There are two approaches to enforcing uniqueness: Either the identifying values are adapted to be unique, yet it is hard to provide a decent strategy to do so and in practice this is rarely desirable. Or the objects which contain duplicate values are merged. We use the latter. It may merge objects, which are not exactly the same, in which case information is lost.

Merging objects along an identifying property comprises two tasks: the objects need to be merged and all associations to these objects need to be updated to point to the merged objects. Both tasks make extensive use of a mapping from original objects to merged objects. As this mapping is computationally complex to derive, we compute it once and reuse the result. The schematical changes for making property `A.f` an identifier are shown in Figure 7. The top-most part shows the table for `A` and associations to this table, which may both be from single-valued associations (columns) as well as multi-valued associations (connection tables). The middle part shows the computed mapping from `A` object ids (source) to merged `A` object ids (target). Only the `target` column has a foreign key to `A*`. At the start of migration the source column also references `A` ids, yet after merge, `source` may point to no longer existing, merged objects. The bottom part shows the schema after migration.

For making `Alias.name` an identifier (step 8 in the running example), Acoda generates the following migration:

```
8 make Alias.name id;
```

```
CREATE TABLE Alias_merge
(  INDEX forward_lookup (source),
   INDEX reverse_lookup (target)   )
   CONSTRAINT `f_Alias_merge`
      FOREIGN KEY `f_Alias_merge`
      (target)
      REFERENCES _Alias (id)
   SELECT original.id AS source, target
   FROM
      _Alias AS original,
      (  SELECT min(id) AS target, Alias_name
         FROM _Alias
         GROUP BY Alias_name    ) AS merged
      WHERE original.Alias_name = merged.Alias_name;
UPDATE _Person AS ref, Alias_merge AS map
   SET ref.Person_alias = map.target
   WHERE ref.Person_alias = map.source;
DELETE FROM _Alias
   WHERE NOT EXISTS
   (  SELECT *
      FROM Alias_merge AS map
      WHERE map.target = id);
ALTER TABLE _Alias
   ADD CONSTRAINT `Alias_name_unique`
   UNIQUE (_name);
DROP TABLE MergeMap_Alias;
```

The first statement computes and stores the mapping form original aliases to merged aliases. It uses two indices for efficient lookup: a forward index to rewrite the associations and a backward index to update the alias table. The second statement updates the `alias` association from `Person`, which at this point in migration is still single-valued. The update joins the `Person` table and the map to update all associations efficiently. The third statement drops the old and redundant aliases, which can now safely be removed, since they are no longer in use. The fourth statement enforces uniqueness and the final statement removes the merge map.

7. Discussion

Related Work. Migration generation is common in software development. Evolving data models require data migration, evolving DTDs require XML migration, and evolving schema require database migration. Furthermore, migration is not restricted to data modeling. It also occurs where meta-models evolve, where domain-specific languages evolve [13], and where grammars evolve [14]. The coupled transformation problem is ubiquitous [11]. In this section, we relate our work to existing work on data model evolution and to work on coupled evolution in general.

Ruby on Rails offers support for migration of databases along an evolving web application[4]. The web applications use an ORM to persist data in a relational database. They offer support for versioning different databases running different versions of the same application. In contrast to our work, the Ruby on Rails migration support requires the developer to define database migrations himself in terms of the relational database. Ruby on Rails offers a set of SQL-like methods to alter a database, such as `create_table`, `add_column` and `remove_index`. They do not offer an evolution language at the application abstraction level.

In the area of data model evolution, most work focuses on evolving schema and migrating databases [2, 5, 6]. Schema describe structure of data storage, primarily focusing on storage techniques to improve query performance. Evolving schema requires the developer to be concerned with database implementation details. In our work, we bridge the ORM to allow the developer to define evolution in the application domain and abstract away from database details. From the application-level evolution specification, we generate schema evolution definitions (in SQL). We rely on the previous work on schema evolution to efficiently map the generated schema evolution onto a database migration. On the one hand, this allows the developer to reason in terms of application logic instead of database techniques. On the other hand, it allows us to introduce more advanced concepts into evolution specifications, such as inheritance, cardinalities, and associations.

Visser et al. formalize the more general coupled transformations [1, 4, 17]: Not only conforming artifacts are considered (such as a database or XML document), also dependent artifact transformations are formalized (such as query and constraint migration). The formalization makes use of data refinement theory and uses Haskell for presentation. Visser et al. do not offer concrete migrations in addition to the presented formalization. Although they

[4] http://guides.rubyonrails.org/migrations.html

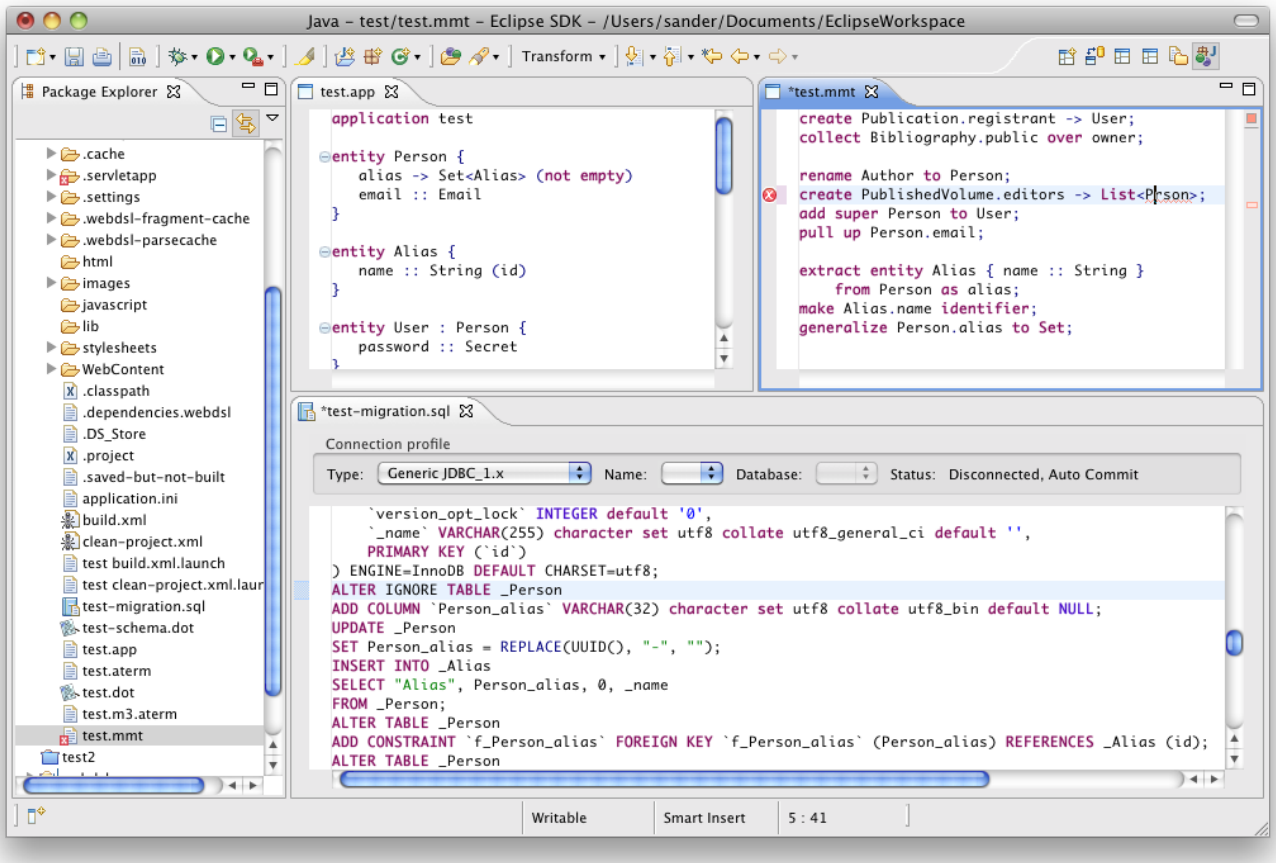

Figure 8. Screenshot of the Acoda Eclipse plugin. The left-most column shows the regular Eclipse project tree. The top-left editor displays a WebDSL data model. The top-right editor shows the evolution specification. The bottom editor shows the generated SQL migration.

consider flattening hierarchies and present a formalization of such, they do not consider inheritance, or a complete ORM. In their concluding remarks, they point out that inheritance would be useful to include, to extend the scope to object-oriented data models.

Lämmel and Lohmann discuss migration of XML data along evolving DTDs [12]. They formalize the migration concepts and distinguish two groups of evolution: refactorings and structure-extending and -reducing evolutions. They discuss higher-level evolutions, such as folding and generalization. Lämmel and Lohmann do not offer a language for describing evolution.

Similar to the application models considered in our work, meta-models are defined in terms of high-level concepts, such as inheritance and cardinalities. Meta-model evolution languages cover a high level of abstraction and are similar to evolution steps on object-oriented data models [3, 7, 9, 18]. We therefore reused the evolution steps defined on meta-models, which are outlined in previous work [8]. In contrast to our work, in meta-modeling, there is a close relationship between the data set structure and the data definition: models closely follow the structure defined in their meta-model. The relational structure of a RDBMS, does not closely follow the object-oriented structure of an application-level model. Thus, where model migration does not need to cover the gap between defined structure and implemented structure, our work covers the mapping between object domain and relational domain: the ORM.

Implementation. The presented evolution modeling language is implemented as a part of Acoda[5]. To seamlessly integrate into regular development, Acoda offers an Eclipse plugin developed using Spoofax/IMP [10]. It operates in cooperation with the (already available) WebDSL plugin, which provides WebDSL application editing and compilation services. Acoda offers additional functionality around evolving WebDSL data models, such as comparison of original and evolved data model to yield an evolution model [15]; editor support for editing evolution models (such as syntax highlighting, instant error marking and content completion); generation of SQL migration code; and application of migrations to a database. The plugin can be used in the context of agile development, in which it supports a short development - migration - deployment - testing loop. For migration of production databases, Acoda also offers a stand alone version, which can be run on-site or remotely.

Figure 8 shows a screenshot of the plugin. The left-most column shows the regular Eclipse project tree. The top-left editor displays a WebDSL data model (the running example). This editor is provided by the WebDSL plugin. The top-right editor shows the evolution specification used throughout the paper, with a small typo to show evolution validity checking and corresponding error marking. This editor is provided by the Acoda plugin. The bottom editor shows the SQL migration generated by the plugin. Although this migration can be viewed and adapted by the developer, general practice is to apply the evolution specification directly, without examining

[5] http://swerl.tudelft.nl/bin/view/Acoda

SQL code. However, this still generates the SQL migration internally, which is then applied to the database.

Changing Persistence Implementation. WebDSL abstracts over implementation details for persistence. The presented migration generation is aligned to Hibernate. But the WebDSL compiler might change some of the parameters for Hibernate's ORM or might even address another persistence framework. Such changes would be transparent to evolution models, since Acoda abstracts over implementation details for migration and preserves WebDSL's data modeling abstractions. The Acoda compiler needs to reflect these changes and has to address the same ORM as the WebDSL compiler. These changes primarily amount to naming differences (of columns, tables, and keys) and a different type of inheritance representation (e.g. using separate tables for each entity, instead of hierarchy tables). To cope with naming differences, the naming in Acoda is pluggable and can be replaced by another naming scheme. To cope with a different inheritance representation, migration generations dealing with inheritance (e.g. the presented sub entity creation, property pull-up, and super addition) need to be adapted. Considering inheritance flattening is the more complex variant, adaptation will generally simplify migration generation.

Performance & Uptime. Databases serve live web applications. Database migration may cause application downtime. Good performance of migration is important to limit downtime.

Acoda constructs migrations from database operations. Efficiency of their implementation depends on the used RDBMS. Nevertheless, we optimize the usage of database operations at two levels: First, we combine evolution operators at the data model level to form more complex operators with a more efficient migration at the database level. For example, a class creation and a feature addition can be combined into a single class creation. Second, we combine SQL operations in the generated migrations at the database level. Acoda compiles a sequence of evolution operators into a sequence of SQL statement sequences. These sequences may overlap. For example, two changes on a table (e.g. a rename and a column addition) may be generated for different evolution operators, yet can be combined into a single `ALTER TABLE` statement, thus significantly improving performance. The two kinds of optimizations target to generate the most efficient migration script.

Furthermore, the generated migrations attempt to shorten the time in which the database is inaccessible as much as possible. For example, the super addition postpones data deletion to the last step, even though it could have been applied earlier. This allows the database to stay online in read-only mode while the more computation intensive steps are executed (such as copying data). Additionally, migrations generally only target a part of the database, remaining application data stays accessible (both readable and writable). In practice however, most migrations are short and can be executed while the application is updated. They cause little or no additional downtime on regular-sized (WebDSL) databases.

Acknowledgments

This research was supported by NWO/JACQUARD project 638. 001.610, *MoDSE: Model-Driven Software Evolution*.

References

[1] T. Alves, P. Silva, and J. Visser. Constraint-aware schema transformation. In *Ninth International Workshop on Rule-Based Programming*, 2008.

[2] P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled schema transformation and data conversion for XML and SQL. In *Practical Aspects of Declarative Languages (PADL 2007)*, volume 4354 of *LNCS*, pages 290–304. Springer, 2007.

[3] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference (EDOC 2008)*. IEEE, 2008.

[4] A. Cunha, J. Oliveira, and J. Visser. Type-safe two-level data transformation. In *Formal Methods Europe (FME 2006)*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006.

[5] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *International conference on management of data (SIGMOD 1993)*, pages 157–166, New York, NY, USA, 1993. ACM.

[6] J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Schema transformation techniques for database reverse engineering. In *Proceedings of the 12th Intl. Conf. on the Entity-Relationship Approach (ER 1993)*, pages 364–375, London, UK, 1994. Springer-Verlag.

[7] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In *ECOOP 2009 - Object-Oriented Programming*. Springer, 2009.

[8] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Software Language Engineering, Third International Conference (SLE 2010)*, LNCS. Springer, 2010.

[9] J. Hoßler, M. Soden, and H. Eichler. Coevolution of models, metamodels and transformations. In *Models and Human Reasoning*, pages 129–154, Berlin, 2005. Wissenschaft und Technik Verlag.

[10] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Domain-specific languages for composable editor plugins. In *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, April 2009.

[11] R. Lämmel. Coupled software transformations (extended abstract). In *First International Workshop on Software Evolution Transformations*, Nov. 2004.

[12] R. Lämmel and W. Lohmann. Format evolution. In *RETIS 01: Proc. 7th International Conference on Reverse Engineering for Information Systems*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

[13] M. Pizka and E. Juergens. Tool supported multi level language evolution. In *In Proceedings of SVM'07: Software and Services Variability Management Workshop Concepts, Models and Tools*, 2007.

[14] S. D. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. In *Model Driven Engineering Languages and Systems (Models 2008)*, volume 5301 of *LNCS*, pages 630–644. Springer, 2008.

[15] S. D. Vermolen, G. Wachsmuth, and E. Visser. Reconstructing complex metamodel evolution. In *Software Language Engineering, Fourth International Conference, SLE 2011, Braga, Portugal, Revised Selected Papers*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012. To Appear.

[16] E. Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *LNCS*. Springer, 2008.

[17] J. Visser. Coupled transformation of schemas, documents, queries, and constraints. *Electron. Notes Theor. Comput. Sci.*, 200(3):3–23, 2008. ISSN 1571-0661.

[18] G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *ECOOP 2007 - Object-Oriented Programming*, volume 4609 of *LNCS*, pages 600–624. Springer Berlin / Heidelberg, 2007.