# An Adapter-Based Approach to Adapt Generated SQL in Model-to-Text Transformations to DB Schema Evolution

Jokin García[1], Oscar Díaz[1], Jordi Cabot[2]

[1] Onekin Research Group, University of the Basque Country (UPV/EHU), San Sebastian (Spain)
`(jokin.garcia, oscar.diaz)@ehu.es`,
[2] Atlanmod, Ecolé des Mines de Nantes - INRIA, Nantes (France)
`jordi.cabot@inria.fr`

**Abstract.** *Forward Engineering* advocates for code to be generated dynamically through model-to-text transformations that target a specific platform. In this setting, platform evolution can leave the transformation, and hence the generated code, outdated. This issue is exacerbated by the perpetual beta phenomenon in Web 2.0 platforms where continuous delta releases are a common practice. Here, manual co-evolution becomes cumbersome. This paper looks at how to automate —fully or in part—the synchronization process between the platform and the transformation. To this end, the transformation process is split in two parts: the stable part is coded as a MOFScript transformation whereas the unstable side is isolated through an adapter that is implicitly called by the transformation at generation time. In this way, platform upgrades impact the adapter but leave the transformation untouched. The work focuses on DB schema evolution, and takes *MediaWiki* as a vivid case study. A first case study results in the upfront cost of using the adapter paying off after three releases *MediaWiki* upgrades.

## 1 Introduction

The changing nature of DB schemas has been a constant concern since the inception of DBs. Software consuming data is dependent upon the structures keeping this data, i.e. the DB schema. Such software might refer to relational views [5], data mappings (i.e. describing how data instances of one schema correspond to data instances of another) [15] or application code [4]. But, what if this software is not directly coded but generated? The popularity of Model-Driven Forward Engineering (FE) is making code generation go mainstream. Key gears of this infrastructure are *Model-to-Text (M2T)* transformations as supported by *Acceleo*[3] or *MOFScript*[4]. M2T transformations (hereafter referred to as just "transformations") obtain "the text" (i.e. the software code akin to a target platform)

[3] http://www.eclipse.org/acceleo/
[4] http://modelbased.net/mofscript/

from a model, i.e. an abstract representation of the solution (a.k.a. Platform Independent Model). As any other application, the generated "text" is fragile upon changes on the underlying platform, i.e. "the text" should co-evolve with the platform. Traditionally, this is achieved at "the text" level [4]. FE opens a different way: co-evolving *not* the generated code but the transformation that generates this code. This is the research question we tackle, i.e. how changes in the DB schema can be propagated to the transformation.

We turn the issue of keeping the application and the DB schema in sync, into one of maintaining the consistency between the code generators (i.e. the transformation) and the DB schema. Despite the increasing importance of FE, this issue has been mostly overlooked. This might be due to understanding that traditional co-evolution techniques can be re-used for transformations as well. After all, transformations are just another kind of applications. However, existing strategies for application co-evolution assume that either the generated SQL script is static or the trace for DB changes is known (and hence, can later be replicated in the application) [4]. However, these premises might not hold in our scenario. Rationales follow:

- transformations do not *specify* but *construct* SQL scripts. The SQL script is dynamically generated once references to the input model are resolved. This specificity of transformations makes it necessary to do the adaptation dynamically after the transformation engine has resolved the references but before it prints the result to the output file. As a result, solutions proposed for "static" scenarios are hardly applicable in this context.
- the trace for DB changes might be unknown. The DB schema and the transformation might belong to different organizations (i.e. there exist an *external* dependency from the transformation to the schema). This rules out the possibility of tracking schema upgrades to be later replicated into the transformation. External dependencies are increasingly common with the advent of the Web 2.0, and the promotion of open APIs and open-source platforms such as Content Management Systems (e.g. *Alfresco*), Wiki engines (e.g. *MediaWiki*) or Blog engines (e.g. *WordPress*). Here, the application (i.e. the portal, the wiki or the blog) is developed on top of a DB schema that is provided by a third party (e.g. the *Wikimedia* foundation).

We had to face the aforementioned scenarios ourselves when implementing *WikiWhirl* [10], a Domain-Specific Language (DSL) built on top of *MediaWiki*. *WikiWhirl* is interpreted, i.e. a *WikiWhirl* model (an expression described along the *WikiWhirl* syntax) delivers an SQL script that is enacted. The matter is that this *SQL* script goes along the *MediaWiki* DB schema. If this schema changes, the script might break apart. Since *MediaWiki* is part of the *Wikimedia Foundation*, we do not have control upon what and when *MediaWiki* releases are delivered. And release frequency can be large (the perpetual-beta effect) which introduces a heavy maintenance burden upon *WikiWhirl*. This paper describes how we faced this issue by applying the adapter pattern to transformations. That is, data manipulation requests (i.e. insert, delete, update, select) are re-directed to the adapter during the transformation. The adapter outputs the code according to the latest schema release. In this way, the main source of instability (i.e. schema upgrades) is isolated in the adapter. The paper describes the adapter architecture, a case study and a cost analysis to characterize the settings that can benefit from this approach. We start by framing our work within the literature in DB schema evolution.

## 2 Related Work

This section frames our work within the abundant literature in DB schema evolution. The aim is *not* to provide an exhaustive list but just some representative examples that serve to set the design space (for a recent review refer to [5]). For our purposes, papers on schema co-evolution can be classified along two dimensions: the target artifact to be co-evolved, and the approach selected to do so. As for the former dimension, the target artifact includes data and applications. As for **data**, upgrading the schema certainly forces to ripple those changes to the corresponding data (e.g. tuples). The strategy used in [7] is for designers to express schema mappings in terms of queries, and then use these queries to evolve the data between the DB schemas. On the other hand, **applications** consuming data are dependent upon the structures keeping this data. Applications might include DB views [5], data mappings [15] or application code [4]. Our work focuses on a specific kind of applications: transformations.

The second dimension tackles the mechanisms used to approach the aforementioned co-evolution scenarios. First, **Application Programming Interfaces** (APIs). This technique allows programs to interface with the DB through an external conceptual view (i.e. the API) instead of a logical view (i.e. the SQL data-manipulation language over the DB schema). This technique is investigated in [3] where they introduce an API which aims at providing application programs with a conceptual view of the relational DB. The second technique is the use of **rewriting methods**. Here, the approach is to replace sub-terms of a formula with other terms. In our setting, this formula can stand for a DB view or data mapping expression (a.k.a. Disjunctive Embedded Dependencies). A rewriting algorithm reformulates a query/mapping upon an old schema into an equivalent query/mapping on the new schema. An example is described in [5]. Next, **DB views** are also used to support co-evolution. Views ensure logical data independence whereby the addition or removal of new entities, attributes, or relationships to the conceptual schema should be possible without having to rewrite existing application programs. Unfortunately, solutions based on views cannot support the full range of schema changes [11]. Finally, **wrappers** enclose artifacts to keep the whole ecosystem functioning. Upon upgrades on the DB schema, the artifact to be wrapped can be either the schema itself or the legacy applications. The first approach is illustrated by [4] where the new schema is encapsulated through an API. The wrapper converts all DB requests issued by the legacy application into requests compliant with the new schema. Alternatively, wrappers can be used to encapsulate the legacy applications. In this case, wrappers act as mediators between the application requests and the DB system. Wrappers for reusing legacy software components are discussed in [12] and [13].

$$\begin{tikzcd}
\mathrm{DBS}_{\mathrm{relational}} \arrow[r, leftrightarrow, "R"] \arrow[d, "U\Delta_{\mathrm{relational}}"'] & \mathrm{T}_{\mathrm{M2T}} \arrow[d, "Y\Delta_{\mathrm{M2T}}"'] \\
\mathrm{DBS'}_{\mathrm{relational}} \arrow[r, leftrightarrow, "R"] & \mathrm{T'}_{\mathrm{M2T}}
\end{tikzcd}$$

**Fig. 1.** Artifacts involved in unidirectional co-evolution. *R* denotes the consistency relationship. (Adapted from [1]).

Our approach can be characterized as being wrapper based. Since the DB schema is beyond our grasp, the wrapper necessarily sits upon the legacy application. However and unlike previous studies, we do not wrap the application itself but the generator (i.e. the transformation) so that the resultant application (i.e. the SQL script) is already tuned to the new DB schema. By working at a higher level (i.e. the transformation), we factor out the adaptation strategy to account for the set of applications handled by the transformation. Figure 1 depicts the main artifacts involved: $\mathrm{DBS}_{\mathrm{relational}}$ is the original DB schema; $\mathrm{T}_{\mathrm{M2T}}$ is the original legacy application (i.e., the version of the transformation that coexisted with the original DB); $\mathrm{DBS'}_{\mathrm{relational}}$ is the new DB schema; $\mathrm{T'}_{\mathrm{M2T}}$ is the new target artifact; $\mathrm{U}\mathit{\Delta}_{\mathrm{relation}}$ is the update applied to the original source[5]; $\mathrm{Y}\mathit{\Delta}_{\mathrm{M2T}}$ is the target update resulting from the coevolution.

$\mathrm{U}\mathit{\Delta}_{\mathrm{relation}}$ can be obtained by recording the changes upon $\mathrm{DBS}_{\mathrm{relational}}$ while the user edits it [4]. However, in our setting this is not always possible since the schema belongs to a third party. Alternatively, an update can be computed using a homogeneous artifact comparison operator, which takes an original version of an artifact and its new version, and returns an update connecting the two. This is our approach. Now, we could obtain transformation update (i.e. $Y\Delta_{M2T}$*) out of* $\mathrm{U}\mathit{\Delta}_{\mathrm{relation}}$. We propose transformations to be engineered for evolution. Therefore, a transformation is conceived as a pair (S,N) where S denotes the stable part, and N stands for the no-stable part. The no-stable part is supported through an adapter. Therefore, $Y\Delta_{M2T}$ denotes the update to be conducted in the adapter. Therefore, this work presents an architecture to compute adapter updates (i.e. $Y\Delta_{M2T}$*)* out of differences between DB schema models (i.e. $\mathrm{U}\mathit{\Delta}_{\mathrm{relation}}$). Our vision is for the adapter to be domain-agnostic, and hence, reusable in other domains. We envisage transformation adapters to play a role similar to DBMS drivers. A DBMS driver shelters applications from the heterogeneity of DBMS. The driver translates the application's data queries into commands that the DBMS understands. Likewise, a transformation adapter seeks to isolate the transformation from changes in the DB schema. As long as these changes are domain agnostic (e.g. the way to face attribute rename is domain independent) then, the adapter can be re-used by the community. Our solution is available in *www.onekin.org/downloads/public/Batch_MofscriptAdaptation.rar*

[5] An **update** is a function that takes an artifact as input, and returns the updated artifact as output, e.g. $DBS'_{relational} = U\Delta_{relational}\ (DBS_{relational})$.

## 3 Case Study

```
 1 texttransformation FreeMind2MediaWiki (in ww:"www.onekin.org/wikiwhirl",
 2   in diff: "http://www.eclipse.org/emf/compare/diff/1.1",
 3   in Ecore: "http://www.eclipse.org/emf/2002/Ecore") {
 4   var timestamp : String = "('%Y%m%d%k%i%s')"
 5   var categoryTitle :  String = wikiRes2.title
 6   var userId: String = "1"
 7          [...]
 8    ww.Categorize::categorize_sql(){
 9       println("UPDATE page set page_touched = " + timestamp +
10       " where page_namespace = 14 and page_title = '" + categoryTitle + "';")
11       println("INSERT into recentchanges (rc_timestamp, rc_cur_time, rc_user,
12       rc_user_text, rc_namespace, rc_title, " + "rc_comment, rc_new, rc_cur_id,
13       rc_this_oldid, rc_last_oldid, rc_type, rc_old_len, rc_new_len, rc_deleted)
14       VALUES (" + timestamp + ", " + timestamp + ", " + userId + ", '" + userName +
15       "', " + namespace + ", '" + pageTitle + "', '" + comment + "', 0, pageId, " +
16       lastRevisionId + ", " + pageRevisionId + ", 0, " + pageLen + ", " + pageLen +
17       " - " + categoryTitle.size() + " - " + "[[Categoy:]]".size()  + ", 0);")
18   }     [...]
```

**Fig. 2.** A *WikiWhirl* transformation snippet. The transformation constructs the SQL statements dynamically from the input model, i.e. the *WikiWhirl* expression (e.g. *categortyTitle* in line 10, *pageTitle* in line 15, etc)

This section outlines the *WikiWhirl* project [9] and the challenges posed by its external dependency with the *MediaWiki* DB schema. Wikis are main exponents of collaborative editing where users join forces towards a common goal (e.g. writing an article about a topic). It comes as no surprise that wikis promote an article-based view as opposed to a holistic view of the wiki content. As a result, APIs and GUIs of wiki engines favor operations upon single articles (e.g. editing and discussing the article's content) while overlooking operations on the wiki as a whole (e.g. rendering the wiki's structure or acting upon this structure by splitting, merging or re-categorizing articles). To amend this, *WikiWhirl* abstracts wiki structure in terms of mindmaps, where refactoring operations (*WikiWhirl* expressions) are expressed as reshapes of mindmaps. Since wikis end up being supported as DBs, *WikiWhirl* expressions are transformed into SQL scripts. Figure 2 shows a snippet of a *WikiWhirl* transformation: a sequence of SQL statements with interspersed dynamic parts that query the input model (i.e. the *WikiWhirl* expression). These statements built upon the DB schema of *MediaWiki*, and in so doing, create an external dependency of *WikiWhirl* w.r.t. *MediaWiki*. In a $4\frac{1}{2}$ year period, the *MediaWiki* DB had 171 schema versions [6]. According to [6], the number of tables has increased from 17 to 34 (100% increase), and the number of columns from 100 to 242 (142%). This begs the question of how to make *WikiWhirl* coevolve with the *MediaWiki* upgrades. Table 1 compares four options in terms of the involvement (i.e. time and focus) and the required technical skills.

**Option 1: manually changing the generated code.** The designer detects that the new release impacts the generated code, and manually updates this code. This approach is discouraged in Forward Engineering outside the protected areas, since subsequent regenerations can override the manual changes. On the upside,

| **Approach** | **Involvement** | **Skills** | **Infrastructure skills** |
|---|---|---|---|
| manually changing the generated code | High | SQL | default |
| manually changing the transformation | High | SQL, MOFScript | default |
| automatically changing the transformation | Low | SQL, ATL | ATL Injectors, ATL metamodel, SQL schema Injectors, Model differentiation |
| automatically *adapting* the transformation | Low | SQL | SQL schema Injectors, Model differentiation |

**Table 1.** Alternatives to manage platform evolution.

this approach acts directly on the generated code, so only SQL skills are required to accomplish the change.

**Option 2: manually changing the transformation.** For sporadic and small transformations, this might be the most realistic option. However, frequent platform releases and/or SQL-intensive transformations make this manual approach too cumbersome and error-prone to be conducted in a regular basis. The user needs to know both the platform language (e.g. SQL) and the transformation language (e.g. *MOFScript*). No additional infrastructure is introduced.

**Option 3: automatically changing the transformation.** The idea is to inject the transformation into a model and next, use a *Higher-Order Transformation* (HOT) [14] to upgrade it. HOTs are used to cater for transformation variability in Software Product Lines [8]. Variability is sought to generate code for different platforms, different QoS requirements, language variations, or target frameworks. The approach is to define "aspects" (i.e. variability realizations) as higher-order transformations, i.e. a transformation whose subject matter is another transformation. Using aspect terminology, the *pointcuts* denote places in a transformation that may be modified by the high-order transformation (HOT). *Advices* modify base transformations through additional transformation code to be inserted before, after, or to replace existing code. Likewise, we could rephrase schema co-evolution as a "variability-in-time" issue, and use HOTs to isolate schema upgrades. Unfortunately, the use of HOTs requires of additional infrastructure: (1) a metamodel for the transformation language at hand, and (2) the appropriate injector/extractor to map from the *MOFScript* code to the *MOFScript* model, and vice versa. The availability of these tools is not always guarantee. For instance, *MOFScript* has both an injector and an extractor. However, *Acceleo* lacks the extractor. Another drawback is generality. It could be possible to develop a HOT for the specific case of *WikiWhirl*. But we aim at the solution to be domain-agnostic, i.e. applicable to no matter the domain metamodel. Unfortunately, HOTs find difficulties in resolving references to the base transformation's input model where its metamodel is unknown at compile time (i.e. where the HOTs is enacted). Moreover, this approach requires additional infrastructure: (1) SQL schema injectors that obtain a model out of the textual description of the DB schema and (2), a model differentiation tool that spots the differences among two schema models. This infrastructure is domain-independent.

**Option 4: automatically adapting the transformation.** The transformation is engineered for change, i.e. variable concerns (i.e SQL scripts) are moved outside the transformation into the adapter. The user does not need to look at the transformation (which is kept unchanged) but at the generated code. SQL skills are sufficient. At runtime, the transformation invokes the adapter instead of directly generating the DB code (i.e. the SQL script). This brings two important advantages. First, and unlike the HOT option, the issue of resolving references to the base transformation's input model, does not exist, as references are resolved at runtime by the transformation itself. Second, this solution does not require the model representation of the transformation (i.e. injectors for the transformation are not needed). On the other side, this approach requires, as option 3, SQL schema injectors and a model differentiation tool.

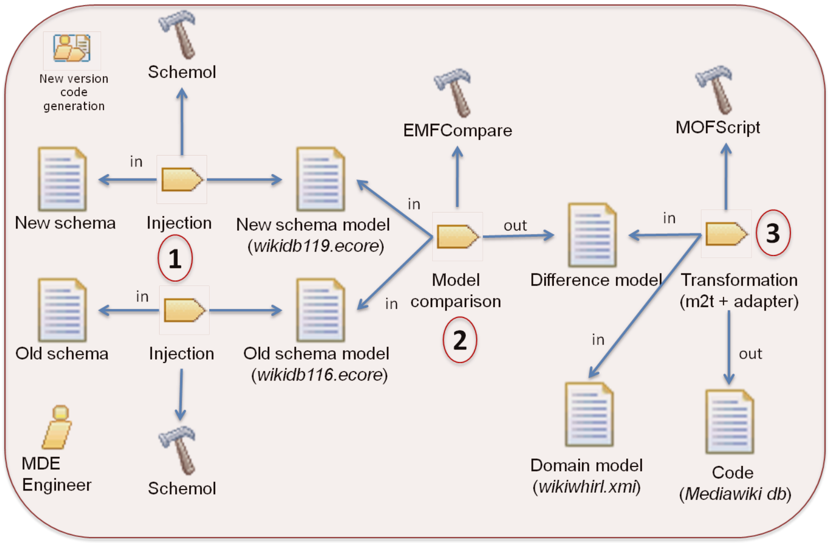
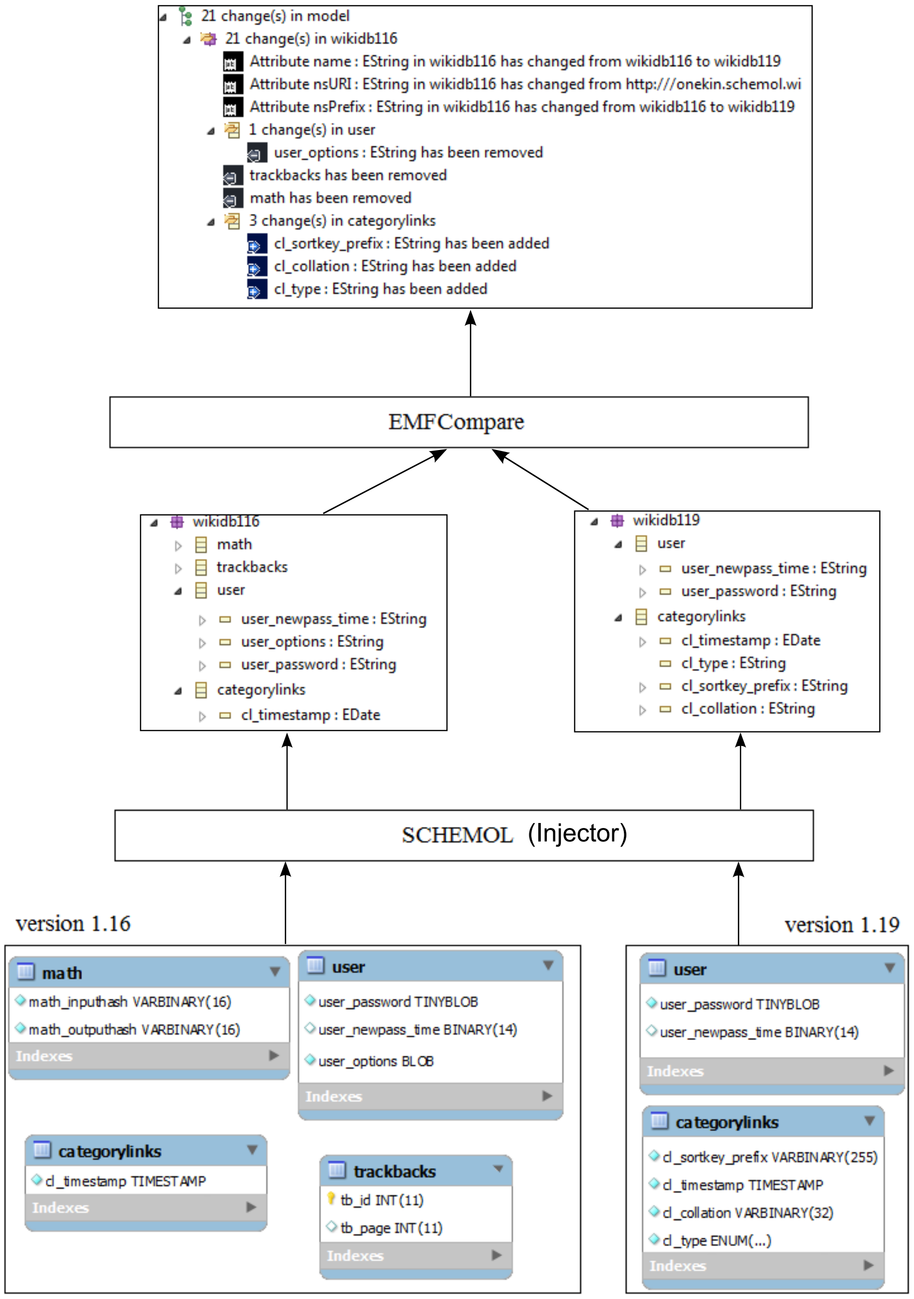

New version code generation
Schemol
EMFCompare
MOFScript
in
New schema
Injection
New schema model (*wikidb119.ecore*)
1
in
out
in
3
Model comparison
Difference model
Transformation (m2t + adapter)
in
Old schema
Injection
Old schema model (*wikidb116.ecore*)
in
2
in
out
MDE Engineer
Schemol
Domain model (*wikiwhirl.xmi*)
Code (*Mediawiki db*)

**Fig. 3.** A generic co-evolution process, exemplified for the WikiWhirl case study.

Figure 3 outlines the different steps of our proposal. First, DB schemas (i.e. *New schema*, *Old Schema*) are injected as *Ecore* artifacts (step 1); next, the schema difference is computed (i.e. *Difference model*) (step 2); finally, this schema difference feeds the adapter used by the transformation (i.e. *MOFScript program*). *MOFScript* is a template-based code generator that uses *print* statements to generate code and language instructions to retrieve model elements. Our approach mainly consists of replacing the *print* statements with invocations to the adapter (e.g. *printSQL*). On the invocation, the adapter checks whether the <*SQL statement*> acts upon a table that is being subject to change. If so, the adapter returns a piece of SQL code compliant with the new DB schema.


```
21 change(s) in model
  21 change(s) in wikidb116
    Attribute name : EString in wikidb116 has changed from wikidb116 to wikidb119
    Attribute nsURI : EString in wikidb116 has changed from http:///onekin.schemol.wi
    Attribute nsPrefix : EString in wikidb116 has changed from wikidb116 to wikidb119
    1 change(s) in user
      user_options : EString has been removed
    trackbacks has been removed
    math has been removed
    3 change(s) in categorylinks
      cl_sortkey_prefix : EString has been added
      cl_collation : EString has been added
      cl_type : EString has been added
```

```
wikidb116
  math
  trackbacks
  user
    user_newpass_time : EString
    user_options : EString
    user_password : EString
  categorylinks
    cl_timestamp : EDate
```

```
wikidb119
  user
    user_newpass_time : EString
    user_password : EString
  categorylinks
    cl_timestamp : EDate
    cl_type : EString
    cl_sortkey_prefix : EString
    cl_collation : EString
```

**Fig. 4.** Injection: from catalog tuples (those keeping the DB schema) to the Difference model.

## 4 Change Detection

Upgrades on the MediaWiki's DB schema are well documented [6]. Developers can directly access this documentation to spot the changes. Gearing towards automatization, these changes can also be ascertained by installing the new release, and comparing the old DB schema and the new DB schema (see Figure 4). The process starts by a notification of a new *MediaWiki* release (e.g. version 1.19). The developer obtains the model for the new schema (*wikidb119*) as well as the model of the schema used in the current release of *WikiWhirl* (*wikidb116*) using some schema injector (e.g. Schemol). Next, schema differences are computed as model differences (e.g. using *EMFCompare).* The output is the *Difference* model.

The *Difference* model is described as a sequence of DB operators. Curino et al. proved that a set of eleven *Schema Modification Operators (SMO)* can completely describe a complex schema evolution scenario. Table 2 indicates the frequency of these change for the *MediaWiki* case, elaborated from [6]. Fortunately, most frequent changes (e.g. *'create table', 'add column', 'drop column'* or *'rename column'*) can be identified from schema differences. Complex changes (e.g. *'distribute table'* or *'merge table'*) cannot be automatically detected and therefore are not included in the table. This kind of changes tend to be scarce. For *MediaWiki*, *'distribute table'* never occurred while *'merge table'* accounts for 1,5% of the total changes.

| **SMO** | **% of usage** | **Change type** | **Adaptation** |
|---|---|---|---|
| Create table | 8.9 | NBC | New comment in the transformation on the existence of this table in the new version |
| Drop table | 3.3 | BRC | Delete statement associated to the table |
| Rename table | 1.1 | BRC | Update name |
| Copy table | 2.2 | NBC | (None) |
| Add column | 38.7 | NBC/ BRC | For *insert* statements: if the attribute is *Not Null*, add the new column in the statement with a default value (from the DB if there is available or according to the type if there is not) |
| Drop column | 26.4 | BRC | Delete the column and value in the statement |
| Rename column | 16 | BRC | Update name |
| Copy column | 0.4 | BRC | Like *add column* case |
| Move column | 1.5 | BRC | Like *drop column* + *add column* cases |

**Table 2.** *Schema Modification Operators* and their adaptation action counterparts.

## 5 Change propagation

Schema changes need to be propagated to the generated code through the transformation. The transformation delegates to the adapter how the SQL command

[6] http://www.mediawiki.org/wiki/Manual:Database_layout

ends up being supported in the current DB schema. That is, *MOFScript's print* is turned into the adapter's *printSQL* (e.g. *"printSQL (<SQL statement>)"*). On invocation, the adapter checks whether the SQL statement acts upon a table that is subject to change (i.e. appears in the *Difference* model). If so, the adapter proposes an adaptation action to restore the consistency. This adaptation action depends on the kind of change. Similar to other co-transformation approaches (e.g. [2]), changes are classified as *(i) Non Breaking Changes (NBC)*, i.e., changes that do not affect the transformation; *Breaking and Resolvable Changes (BRC)*, i.e., changes after which the transformations can be automatically co-evolved; and *Breaking and Unresolvable Changes (BUC)*, i.e., changes that require human intervention to co-evolve the transformation. Based on this classification, different contingency actions are undertaken: no action for NBC, automatic co-evolution for BRC, and assisting the user for BUC. Table 2 describes this typology for DB changes, the usage percentage of each change for *MediaWiki* [6], and the adaptation counterpart.

```
1  printSQL(statement: String){
2    [...]
3     var tableName : String = java("org.gibello.zql.ZqlParser",
4     "getTableName", statement , CLASSPATH );
5     diff.objectsOfType(diff.RemoveModelElement)->forEach
6     (rme:diff.RemoveModelElement | rme.rightParent.name=tableName){
7         var paramsRemoveColumn:List;
8         paramsRemoveColumn.add(statement);
9         paramsRemoveColumn.add(rme.rightParent.name);
10        paramsRemoveColumn.add(rme.leftElement.name);
11        println("#"+statement);
12        println(java("org.gibello.zql.ZqlParser", "removeColumn",
13        paramsRemoveColumn, CLASSPATH) + ";");
14    }
15    [...]
16 }
```

**Fig. 5.** The adapter. A *printSQL* function that handles the *"remove column"* case.

Implementation wise, the adapter has two inputs: the *Difference* model and the model for the new schema (to obtain the full description of new attributes, if applicable). The ZQL open-source SQL parser was used[7] to parse SQL statements to Java structures. This parser was extended to account for adaptation functions to modify the statements (e.g. *removeColumn*) and support functions (e.g. *getTableName*). Figure 5 provides a glimpse of the adapter for the case *"remove column"*. It starts by iterating over the changes reported in the *Difference* model (line 5). Next, it checks (line 6) that the deleted column's table corresponds with the table name of the statement (retrieved in lines 3-4). Then, all, the statement, the table name and the removed column are added to a list of parameters (lines 7-10). Finally, the adapter outputs an SQL statement without the removed column, using a function with the list of parameters that modifies the expression (lines 12-13). The adaptation process is enacted for each *SQLprint* statement, regardless of whether the very same statement has been previously

[7] http://zql.sourceforge.net/

```
 1  INSERT INTO categorylinks (cl_from, cl_to, cl_sortkey, cl_timestamp)  VALUES
 2    (@pageId, 'Softwareproject', 'House_Testing',
 3    (DATE_FORMAT(CURRENT_TIMESTAMP(), '%Y%m%d%k%i%s'));
 4  INSERT INTO trackbacks (tb_name, tb_title, tb_url, tb_ex, tb_id, tb_page)
 5    VALUES ('trackback1', 'title', 'http://blog/post', '', '', '');          version 1.16
 6  INSERT INTO user (user_id, user_name, user_real_name, user_password,
 7    user_newpassword, user_newpass_time, user_email, user_options,
 8    user_touched, user_token, user_email_token_expires, user_registration,
 9    user_editcount) VALUES ('1', 'Jokin', 'Jokin Garcia', 'c7c1105fac', '', NULL,
10    'jokin.garcia@ehu.es', 'quickbar=1', '20110902', 'd863a16e41', '', '20070718', '1360');
```

```
 1  #WARNING: Added columns cl_type, cl_sortkey_prefix and cl_collation
 2  INSERT INTO categorylinks (cl_from, cl_to, cl_sortkey, cl_timestamp, cl_type,
 3    cl_sortkey_prefix,cl_collation) VALUES (@pageId, 'Softwareproject', 'House_Testing',
 4    (DATE_FORMAT(CURRENT_TIMESTAMP(), '%Y%m%d%k%i%s'), 'page', '', '0');
 5  #WARNING: Deleted table trackbacks
 6  #INSERT INTO trackbacks (tb_name, tb_title, tb_url, tb_ex, tb_id, tb_page)
 7  #VALUES ('trackback1', 'title', 'http://blog/post', '', '', '');
 8  #WARNING: Deleted column user_options in table user                       version 1.19
 9  #INSERT INTO user (user_id, user_name, user_real_name, user_password,
10  #user_newpassword, user_newpass_time, user_email, user_options,
11  #user_touched, user_token, user_email_token_expires, user_registration,
12  #user_editcount) VALUES ('1', 'Jokin', 'Jokin Garcia', 'c7c1105fac', '', NULL
13  #'jokin.garcia@ehu.es', 'quickbar=1', '20110902', 'd863a16e41', '', '20070718', '1360');
14  INSERT INTO user (user_id, user_name, user_real_name, user_password,
15    user_newpassword, user_newpass_time, user_email, user_touched,
16    user_token, user_email_token_expires, user_registration, user_editcount)
17    VALUES ('1', 'Jokin', 'Jokin Garcia', 'c7c1105fac', '', NULL,
18    'jokin.garcia@ehu.es', '20110902', 'd863a16e41', '', '20070718', '1360');
```

**Fig. 6.** MediaWiki 1.16 generated script *versus* MediaWiki 1.19 generated script. Since the MOFScript code keeps constant, differences are due to the adapter. The adapter also intermingles comments (#) to ease user inspection.

processed or not. Though this penalizes efficiency, the frequency and the time at which this process is conducted make efficiency a minor concern.

Back to our sample case, the SQL script in Figure 6 is the result of enacting the generation process with *wikidiff_v16v19* as the *Difference* model. Once references to the variables and the model elements have been resolved, MOFScript's *printSQL* statements invoke the adapter. The adapter checks whether either the tables or the attributes of the *printSQL* statement are affected by the upgrade (as reflected in the *Difference* model) and applies the appropriate adaptation (see table Table 2). Specifically, the *Difference* model *wikidiff_v16v19* reports:

1. the introduction of three new attributes in the *categorylinks* table, namely, *cl_type*, *cl_sortkey_prefix* and *cl_collation*. Accordingly, the adapter generates *SQL* insert/update statements where new attributes which are 'Not Null' are initialized with their default values (lines 1-4 below);
2. the deletion of tables *math* and *trackback*. This causes the affected *printSQL* statements to output nothing (i.e. the old output is left as a comment) (lines 5-7 below);
3. the deletion of attribute *user_options* in the *user* table. Consequently, the affected *printSQL* statements, output the SQL but removing the affected attributes (lines 14-18 below). In addition, a comment is introduced to note this fact (lines 8-13 below).

## 6 Assessment

| Version | #Add column | #Impacts on WikiWhirl | #Drop column | #Impacts on WikiWhirl |
|---|---|---|---|---|
| 1.17 | 1 | 3 | 0 | 0 |
| 1.18 | 0 | 0 | 1 | 1 |
| 1.19 | 2 | 11 | 1 | 0 |

**Table 3.** Co-evolving *WikiWhirl* from *MediaWiki* 1.16 to *MediaWiki* 1.19. Effort estimated in terms of the number of *MOFScript* instructions affected.

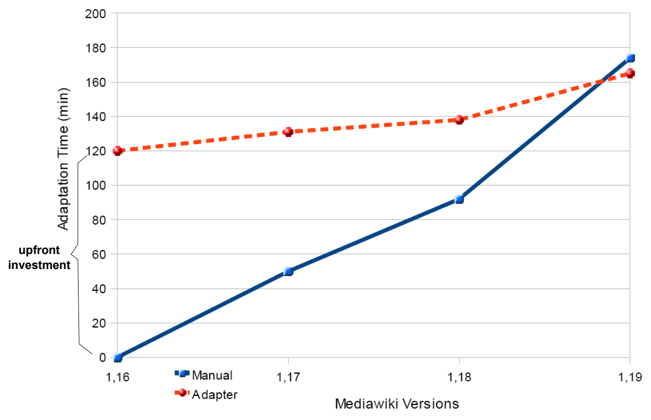


**Fig. 7.** Accumulative costs of keeping *WikiWhirl* and *MediaWiki* in sync. Comparison of the manual (continuous line) and the assisted approach (dotted line).

The net value of an adapter is given by the cost savings that occur to accommodate each DB release minus the development cost of the adapter. These savings are expected from *Breaking and Resolvable Changes (BRC)* since (1) they are amenable to be automated, and (2) they account for the majority of the change. Next paragraphs provide some figures for BRCs, comparing the manual vs the adapter-based approach. To this end, we conducted an assessment on the cost of migrating a *WikiWhirl*'s *MOFScript* transformation from version 1.16 to version 1.17 of the *MediaWiki* DB schema. Table 3 provides some figures of the schema changes and their impact. The experiment was conducted by 8 PhD students who were familiarized with SQL, *MOFScript* and *Ant* [8] (4 for the manual and 4 for the assisted).

### Manual propagation

Subjects conducted two tasks: (1) identifying changes between the two versions from the documentation available in the *MediaWiki* web pages, and (2), adapting manually the transformation. The following equation resumes the main costs:

[8] http://ant.apache.org/

*Manual Cost* = $D + P$ * *#Impacts*

being $D$: the time estimated for detecting whether the new *MediaWiki* release impacts the transformation, $P$: the time needed to **P**ropagate a single change to the *MOFScript* code, and *#Impacts:* the number of instructions in the transformation **I**mpacted by the upgrade.

**The experiment.** $D$ very much depends on the documentation available. For *MediaWiki*, designers should check the website[9], navigate through the hyperlinks, and collect those changes that might impact the code. The experiment outputted an average of 38' for $D_{MediaWiki}$, which is not very high due to the subjects being already familiarized with the *MediaWiki* schema. Next, the designer peers at the code, updates it, and checks the generated code. Subjects were asked to provide a default value for the newly introduced columns. On average, this accounts for 4' for a single update (i.e. $P_{BRC}$ = 4'). Since the 1.17 upgrade impacted 3 *MOFScript* instructions, this leads to a total cost of 50' (i.e. 38 + 4*3). The execution time is considered negligible in both the manual and the assisted options since it is in the order of seconds.

## Assisted propagation

Subjects conducted two tasks: (1) configuration of the batch that launches the assisted adaptation, and (2), verification of the generated SQL script. The batch refers to a macro that installs the new DB release, injects the old schema and new schema, obtains the difference model, and finally, executes the adapter-aware *MOFScript* code. This macro is coded in *Ant* and some shell script commands. Running this macro outputs a *MOFScript* snippet along the lines of the new DB schema. Designers should look at this upgraded code since some manual intervention might still be needed. For instance, the introduction of new columns might also involve the assignment of a value that might not coincide by the one assigned by the adapter. Likewise, column deletion, though not impacting the transformation as such, might spot some need for the data in the removed column to be moved somewhere else. Worth noticing, the designer no longer consults the documentation but relies on the macro to spot the changes in the *MOFScript*. We assume, the comments generated by the adapter are expressive enough for the designer to understand the change (Figure 6). On this basis, the designer has to verify the proposed adaptation is correct, and amend it, if appropriate. The following equation resumes the main costs:

*Assisted Cost* = $C + V$ * *#Impacts*

being $C$: the time needed to **C**onfigure the batch; $V$: the time needed to **V**erify that a single automatically adapted instruction is correct and to alter it, if applicable.

**The experiment.** It took an average of 5' for the subjects to configure the macro (mainly, file paths) to the new DBMS release. As for V, it took an average

[9] http://www.mediawiki.org/wiki/Manual:Database_layout

of 6' for users to check the *MOFScript* code. Therefore, the assisted cost goes up to 11'.

Similar studies where conducted for other *MediaWiki* versions. Figure 7 depicts the accumulative costs of keeping *WikiWhirl* and *MediaWiki* in sync. Actually, till version 1.16 all upgrades were handled manually. Ever since, we resort to the macro to spot the changes directly on the generated *MOFScript* code. Does the effort payoff? Figure 7 shows the breakeven. It should be noted that subjects were already familiarized with the supporting technologies. This might not be the case in other settings. Skill wise, the manual approach is more demanding on *MOFScript* expertise while it does not require *Ant* knowledge. Alternatively, the assisted approach requires some knowledge about *Ant* but users limit themselves to peer at rather than to program *MOFScript* transformations.

The cost reduction rests on the existence of an infrastructure, namely, the adapter and the macro. The adapter is domain-agnostic, and hence, can be reused in other domains. On these grounds, we do not consider the adapter as part of the development effort of the *WikiWhirl* project in the same way that DBMS drivers are not included as part of the cost of application development. However, there is a cost of familiarizing with the tool, that includes the configuration of the batch macro (e.g. DB settings, file paths and the like). We estimated this accounts for 120' (reflected as the upfront investment for the assisted approach in Figure 7). On these grounds, the breakeven is reached after the third release.

## 7 Conclusions and future work

The original contribution of this paper is to address, for a specific case study, the issue of transformation co-evolution upon DB schema upgrades. The suitability of the approach boils down to two main factors: the DB schema stability and the transformation coupling (i.e. the number of SQL instructions in the *MOFScript* code). If the DB schema stability is low (i.e. large number of releases) and the transformation coupling is high, the cost of keeping the transformation in sync, increases sharply. In this scenario, we advocate for a preventive approach where the transformation is engineered for schema stability: *MOFScript's 'print'* is substituted by the adapter's *'printSQL'*. The adapter, using general recovery strategies, turns SQL statements based on the *old* schema into SQL statements based on the *new* schema.

That said, this approach presupposes that the impact of DB schema changes are confined to the SQL statements without affecting the logic of the transformation itself. In our experience, this tends to be the case for medium-size upgrades. However, substantial changes in the DB schema might require changing the transformation logic. This is certainly outside the scope of the adapter. Next follow-on includes to generalize this approach to other settings where schema evolution is also an issue such as ontologies or XML schemas.

## Acknowledgements

This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2011-23839 *(Scriptongue)*. Jokin enjoyed a pre-doctoral grant from the Basque Government under the “Researchers Training Program”.

## References

1. M. Antkiewicz and K. Czarnecki. Design space of heterogeneous synchronization. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 3–46. Springer, 2007.
2. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Enterprise Distributed Object Computing Conference*, 2008.
3. A. Cleve, A.F. Brogneaux, and J.L. Hainaut. A conceptual approach to database applications evolution. In Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson C. Woo, and Yair Wand, editors, *ER*, volume 6412 of *Lecture Notes in Computer Science*, pages 132–145. Springer, 2010.
4. A. Cleve and J. Hainaut. Co-transformations in database applications evolution. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 409–421. Springer, 2005.
5. C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. *VLDB J.*, 22(1):73–98, 2013.
6. C. A. Curino, L. Tanca, H. J. Moon, and C. Zaniolo. Schema evolution in wikipedia: toward a web information system benchmark. In *In ICEIS*, 2008.
7. Renee J. Miller, Mauricio A. Hernandez, Laura M. Haas, Lingling Yan, Howard C. T. Ho, Ronald Fagin, and Lucian Popa. The clio project: managing heterogeneity. *SIGMOD Rec.*, 30(1):78–83, March 2001.
8. J. Oldevik and Ø. Haugen. Higher-order transformations for product lines. In *SPLC*, pages 243–254. IEEE Computer Society, 2007.
9. G. Puente, O. Díaz, and M. Azanza. Refactoring affordances in corporate wikis: a case for the use of mind maps. *Enterprise Information Systems*, 0(0):1–50, 2013.
10. G. Puente and O. Díaz. Wiki refactoring as mind map reshaping. In Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza, editors, *CAiSE*, volume 7328 of *Lecture Notes in Computer Science*, pages 646–661. Springer, 2012.
11. Y. G. Ra. Relational schema evolution for program independency. In *Proceedings of the 7th international conference on Intelligent Information Technology*, CIT'04, pages 273–281, Berlin, Heidelberg, 2004. Springer-Verlag.
12. H. M. Sneed. Encapsulation of legacy software: A technique for reusing legacy software components. *Annals of Software Engineering*, 9(1-4):293–313, 2000.
13. P. Thiran, J.L. Hainaut, G.J. Houben, and D. Benslimane. Wrapper-based evolution of legacy information systems. *ACM Trans. Softw. Eng. Methodol.*, 15(4):329–359, October 2006.
14. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In Richard Paige, Alan Hartman, and Arend Rensink, editors, *ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.
15. Y. Velegrakis, J. Miller, and L. Popa. Preserving mapping consistency under schema changes. *The VLDB Journal*, 13(3):274–293, September 2004.