Inf Syst Front (2018) 20:45–61
DOI 10.1007/s10796-016-9730-2

Robust and simple database evolution

Kai Herrmann[1] · Hannes Voigt[1] · Jonas Rausch[1] · Andreas Behrend[2] · Wolfgang Lehner[1]

Published online: 24 January 2017

Abstract Software developers adapt to the fast-moving nature of software systems with agile development techniques. However, database developers lack the tools and concepts to keep the pace. Whenever the current database schema is evolved, the already existing data needs to be evolved as well. This is usually realized with manually written SQL scripts, which is error-prone and explains significant costs in software projects. A promising solution are declarative database evolution languages, which couple both schema and data evolution into intuitive operations. Existing database evolution languages focus on usability but do not strive for completeness. However, this is an inevitable prerequisite to avoid complex and error-prone workarounds. We present CoDEL which is based on an existing language but is relationally complete. We precisely define its semantic using relational algebra, propose a syntax, and formally validate its relational completeness. Having a complete and comprehensive database evolution language facilitates valuable support throughout the whole evolution of a database. As an instance, we present VaCo, a tool supporting developers with variant co-evolution. Given a variant schema derived from a core schema, VaCo uses the richer semantics of CoDEL to semi-automatically co-evolve this variant with the core.

Keywords Database evolution · Evolution language · Relational completeness · Co-Evolution

1 Introduction

Changes in modern software systems are no longer an exception but have become daily business. Following the mantra "Evolution instead of Revolution", agile software development focuses the creativity and excellence of people to handle the unpredictably dynamic world of software development (Beck et al. 2001). Agile methods are characterized by short development cycles, each with the goal of a shippable product. This creates constant feedback, which helps to establish a customer-oriented development process resulting in products that fit customers' true needs and yield high customer acceptance. It is in the very nature of agile development, that requirement specifications are in perpetual flux. Adjusting the software's design to updated requirements is as much daily business as developing new features.

To handle complexity, modern software systems are composed of a multitude of single components, e.g. core components, common extensions, customized extensions, and respective variants, which can all be developed by different teams (Herrmann et al. 2015). However, each single component is, at the same time, exposed to the discussed agile

✉ Andreas Behrend
behrend@cs.uni-bonn.de

Kai Herrmann
kai.herrmann@tu-dresden.de

Hannes Voigt
hannes.voigt@tu-dresden.de

Jonas Rausch
jonas.rausch@tu-dresden.de

Wolfgang Lehner
wolfgang.lehner@tu-dresden.de

[1] Dresden Database Systems Group, Technische Universität Dresden, Dresden, Germany

[2] Computer Science III, University of Bonn, Bonn, Germany

evolution process, which, in turn, adds significant complexity to the evolution of the whole system. The orchestrated co-evolution of all components is a major challenge in software development. While the co-evolution is straight forward for independent components interacting over well defined interfaces, it becomes a challenge for depending components, like variants or (customized) extensions of a given core component.

A major obstacle in the whole process of evolution and co-evolution are the database systems (Ambler and Sadalage 2006). Whereas software development tools support developers in the process of both designing changes with a comprehensive set of automatized refactoring features and (semi-) automated co-evolution, the evolution of databases is usually realized by manually writing scripts of SQL-DDL and -DML operations. Due to the separation of schema changes (DDL) and the actual data migration (DML), the developer's intention is lost between the lines. Any subsequent development task like ensuring consistency, co-evolving other components, reducing downtime, optimizing the migration, as well as documentation needs to be performed additionally and manually. This manual database evolution is expensive and error-prone, particularly because many software projects show poor integration of the database developers. According to a survey (Ambler 2006), two third of the polled software developers perform database-related changes without consulting the responsible database developers, which certainly increases the software developer's productivity but is not necessarily increasing the quality of the resulting database.

To keep pace with agile software development, the database systems have to supply software-refactoring-like features. Such database evolution features need to evolve the database schema (schema evolution) and payload data (data evolution) in a single consistent step (Roddick 1995). The proposed database evolution process is illustrated in Fig. 1. While evolving an application, the developer specifies the corresponding database evolution with the help of *schema modification operations* (SMOs). In contrast to SQL-DDL and -DML statements, SMOs specify the evolution of the schema and the data in a descriptive, integrated way and ensure that the data is consistently evolved with the schema. SMOs are typically more compact than a script of DDL and DML operations resulting in the same evolution. On the user side, SMOs increase the developer's productivity while dealing with database evolution and reducing the chances of faulty evolution scripts and unintended data loss. As SMOs capture the developer's intention, they can be e.g. combined in a meaningful way greatly simplifying the co-evolution of components/variants. On the DBMS side, SMOs open the opportunity to optimize and reduce the data movement in an evolution step or even invert evolution steps for database versioning with co-existing schema versions (Herrmann et al. 2016). These benefits are enabled by using SMOs instead of DDL/DML. A set of SMOs forms a *database evolution language*.

Naturally, the design of a database evolution language determines its expressiveness. A powerful database evolution language lets the user easily specify all necessary evolution steps. In contrast, a weak one forces the user into more complicated evolution scripts or even to fall back on DDL/DML statements, which renders the database evolution language useless. A database evolution language should at least cover the power of DDL and DML of a standard database system. We argue, that a database evolution language for relational databases should at least be *relational complete*: for any relational DDL/DML script, there exists a semantically equivalent sequence of SMOs. Relational DDL/DML scripts create, alter, and drop database objects, while conditions and the actual data are specified using DQL expressions. The latter motivates the relational algebra (Codd 1970) as the natural reference for the power of relational DDL and DML. We present CoDEL, a relational complete database evolution language with precisely defined syntax and semantics (Herrmann et al. 2015). Developers can purely rely on CoDEL to specify any database evolution. This makes CoDEL a great facilitator for further tool support for database evolution in order to catch up on the agile software development (Herrmann et al. 2016). In this article, we focus on the relational completeness of CoDEL

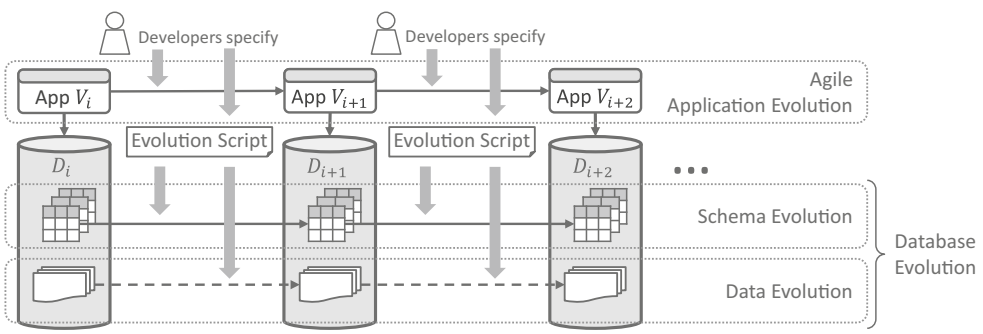


Fig. 1 Database evolution

and show one example of great tool support facilitated by CoDEL: semi-automated variant co-evolution. Regarding the expressiveness of CoDEL, we intentionally exclude the evolution of constraints. In practice, constraint evolution is a very relevant aspect, which has already been solved conveniently for PRISM++ (Curino et al. 2010). Since this solution is applicable to CoDEL as well, we do not cover constraint evolution here again.

Example Let us consider the example shown in Fig. 2 to illustrate the power of declarative database evolution. We develop an application *TasKy* to manage tasks each having a priority, ranging from 3 (lowest) to 1 (highest), as well as an associated author. The first release of *TasKy* stores all its data in the two tables Task(id,task,prio,author) and Author(id,name) where author is a foreign key from Task to the id of Author. After productive go-live, users add data to *TasKy* as shown in this figure.

Starting with this core application we will derive a variant for the mobile application *Do!* to manage short term todos at any place and any time. *Do!* has three major differences to *TasKy*. First, we only show the most urgent tasks with the highest priority and store the remaining tasks in an archive, which needs to be accessed separately. Second, we eliminate the priority column from the short term todos, since it is always 1 for those tasks. Third, we additionally need a new column to store whether the todos have been recognized in the phones notification bar or not. We obtain this variant with the CoDEL script:

```
EVOLUTION FROM 'TasKy' TO 'Do!':
PARTITION TABLE Task INTO Todos WITH prio=1 AND Archive WITH prio>1;
DROP COLUMN prio FROM Todos;
ADD COLUMN new AS 'True' INTO Todos;
```

This script contains all required information to create the new schema and migrate the existing data while still capturing the developer's intention explicitly. We continue the development of the core *TasKy* to a new version *TasKy2*. Among other improvements, this update includes a denormalization as we join the tasks and the authors. With CoDEL, we simply write:

```
EVOLUTION FROM 'TasKy' TO 'TasKy2':
JOIN TABLE Task AND Author INTO Task ON Task.author=Author.id;
DROP COLUMN author FROM Task;
RENAME COLUMN name IN Task TO author;
```

Again, this script carries enough information to allow creating the new schema and transforming the existing data accordingly. However, there is a problem now: the variant *Do!* is based on *TasKy*, which has now evolved to *TasKy2* and does not exist any longer. In (Herrmann et al. 2016) we present InVerDa, a tool to generate co-existing schema versions from CoDEL-like evolutions. InVerDa keeps old schema versions alive, both for reading and writing, and hence could keep both *TasKy* and the initial variant *Do!* accessible.

However, DBMSes usually do not have such advanced features and it is more feasible to simply co-evolve the variant matching it to the new core version. We derive a new variant *DoTwo!*, which reflects the intentions of the initial variant *Do!* but is derived from *TasKy2*. Manually co-evolving the variant is an expensive and complex task in practice. Fortunately, it pays off now to have both the core evolution and the variant defined as CoDEL scripts. We present VaCo a tool for semi-automatic co-evolution of a variant with its core. As CoDEL evolution scripts precisely capture the developer's intention, VaCo can semi-automatically propose an evolution from *TasKy2*

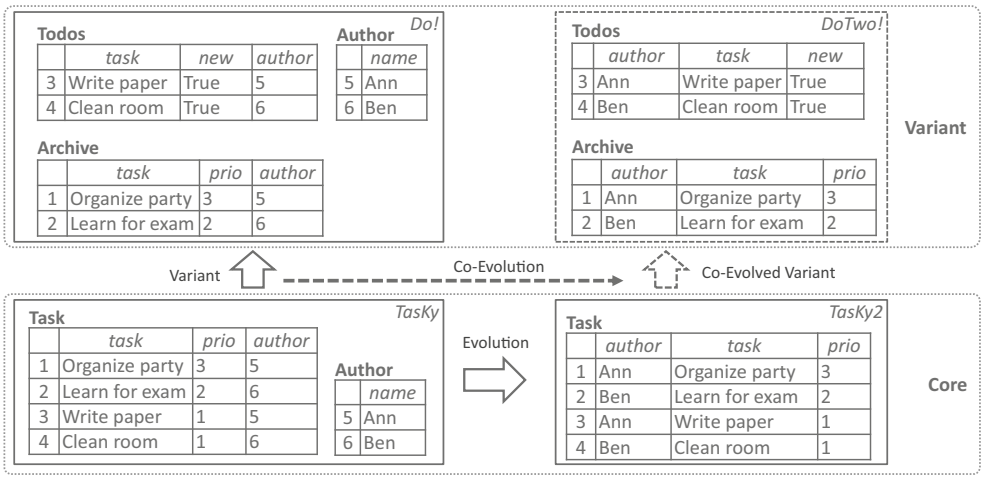


Fig. 2 Example for Database Evolution and the Co-Evolution of the Variant

to the co-evolved variant *DoTwo!*. The developer merely has to give simple decisions for ambiguous combinations. In our example, VACO would automatically propose:

```
EVOLUTION FROM 'TasKy2' TO 'DoTwo!':
PARTITION TABLE Task INTO Todos WITH prio=1 AND Archive
WITH prio>1;
DROP COLUMN prio FROM Todos;
ADD COLUMN prio AS 'True' INTO Todos;
```

The new variant combines both the intent of the core evolution and the variant. Through the simple and intuitive format of CODEL the developer can easily understand and modify the proposed sequence of SMOs in case the developer intends to do further adjustments to the proposed *DoTwo!*.

Declarative database evolution languages like CODEL enable sophisticated tool support throughout the whole process of database evolution. Using the relational complete CODEL, developers can specify any evolution at hand. So, CODEL does not only simplify database evolution and makes it more robust but also opens many opportunities to automate further tasks associated to database evolution. VACO is a vivid example for the power of CODEL, as it realizes semi-automated variant co-evolution, a repetitive, expensive, and error-prone task. Our contributions are:

1. We provide formal definitions of the **semantics** of CODEL's operations and an SQL-like syntax. With that, CODEL can serve as a reference language for the formal evaluation of other database evolution languages.
2. We formally validate the **relational completeness** of CODEL. We show that all operations of the relational algebra can be expressed with CODEL.
3. We realize **semi-automated variant co-evolution** based on CODEL. Given a core schema from which we derive a variant, it is possible to co-evolve the variant whenever the core evolves.
4. CODEL is relationally complete with precisely defined semantics. So, it is a perfect basis for **further research toward agile database evolution**. Researchers can "Divide and Conquer" the challenges on a per-SMO-basis.

We define our database evolution language CODEL in Section 2. We show that CODEL can handle common evolution scenarios just like PRISM and additionally prove its relational completeness in Section 3. With that, CODEL facilitates valuable features for supporting database evolution. In Section 4, we present semi-automatic co-evolution of variants as an example. Finally, we discuss related work in Section 5 and conclude the paper in Section 6.

2 CODEL

CODEL is a relational complete database evolution language. We introduce both a minimal relational complete database evolution language $\mathcal{L}_{\text{min}}$ and our comprehensive CODEL. By showing the equivalent expressiveness of these two languages, we formally validate CODEL's relational completeness. In general, a database evolution language $\mathcal{L}$ is a set of SMOs with parameters to be instantiated. For instance, the SMO to drop a column from a table requires two parameters: the name of the table and the name of the column. Let $inst(\mathcal{L})$ be the set of all operation instances of $\mathcal{L}$ with valid parameters. Then, a relational database $D = \{R_1, \ldots, R_n\}$ with tables R_i can be evolved to another relational database $D' = \{R'_1, \ldots, R'_m\}$ with a sequence of SMOs $s \in inst(\mathcal{L})^+$. We formally denote this as: $D \xrightarrow{s} D'$.

The whole evolution history is a tree-shaped graph with database versions D as nodes—the initial database is the root node—and sequences of SMOs $s \in inst(\mathcal{L})^+$ as edges. A single database exists exclusively in one version, however, the evolution tree captures all the information about the evolution to other versions serving e.g. as documentation or as basis for semi-automated co-evolution as we will show in this work as well.

The **relational completeness** of a database evolution language is a fundamental prerequisite, as the advantages of database evolution languages can only be used when the developer specifies all evolution steps with the given set of SMOs. So, we have to make sure that there is no need for a developer to fall back on traditional SQL in any situation. We take the relational algebra as the key reference for the expressiveness of traditional SQL. A minimal language providing relational completeness is $\mathcal{L}_{\text{min}} = \{\text{ADD}(\cdot,\cdot), \text{DEL}(\cdot)\}$ with

$$\begin{aligned} \text{ADD}\big(R', \epsilon\big) &\rightarrow D \cup \big\{R' = \epsilon(R_1, \ldots, R_n)\big\} \\ \text{DEL}(R) &\rightarrow D \setminus \{R\} \end{aligned}$$

The $\text{ADD}(\cdot,\cdot)$ operation adds a new table R' to the database D based on the given relational algebra expression ϵ that works on the relations of D. The $\text{DEL}(\cdot)$ operation removes the specified table R from D. A database D can be evolved to any other database D' with a sequence $s \in inst(\mathcal{L}_{\text{min}})^+$, where the tables in D' are computed from D with relational algebra expressions in the $\text{ADD}(\cdot,\cdot)$ operation. So, $\mathcal{L}_{\text{min}}$ is relationally complete. From a practical standpoint however, $\mathcal{L}_{\text{min}}$ is not very appealing, because it is rather unintuitive and not oriented on actual evolution steps. However, any other database evolution language that is as expressive as $\mathcal{L}_{\text{min}}$ is relationally complete as well.

To the best of our knowledge, one of the most advanced database evolution language designs is PRISM++ (Curino et al. 2008; Curino et al. 2010). PRISM++ provides SMOs to create, rename, and drop both tables and columns, to divide and combine tables both horizontally and vertically, and to copy tables. The PRISM++ authors claim practical completeness for their powerful database evolution language, by validating it against evolution histories of several open source projects. Although this evaluation suggests that PRISM++ is sufficient also for other software projects, it does not provide any reliable completeness guarantee. For instance, we do not see an intuitive way to remove all rows from a table A, which also occur in a table B using the PRISM++ database evolution language, since it does not offer any direct or indirect outer join functionality. Thus, we consider PRISM++ not to be relationally complete. Nevertheless, PRISM++ has an intuitive and field-proven design. In this work, we present a relational *Complete Database Evolution Language* (CoDEL), inspired by the set of PRISM++ SMOs to inherit its practical feasibility. However, CoDEL is relationally complete and equally expressive as $\mathcal{L}_{\text{min}}$.

Database evolution changes the schema of a database and/or the already existing data. A database evolution language contains operations to descriptively specify such changes as units, which clearly distinguishes it from SQL-DDL and -DML. PRISM++ limits itself to operations that modify individual tables – no PRISM++ operation accepts more than two tables. This keeps the PRISM++ database evolution language intuitive and easy to learn. CoDEL adopts this principle. However, CoDEL operations systematically cover all possible changes that can be applied to tables. Tables are the fundamental structuring element and the container for payload data in a relational database. Secondary database objects such as views, constraints, functions, stored procedures, indexes, etc. should be considered in database evolution as well. However, in this paper we focus on the evolution of the primary data.

Our **database evolution language CoDEL** defines SMOs of the pattern $\langle smo \rangle_{\langle scope \rangle}(\Theta)$, where $\langle smo \rangle$ is the type of operation, $\langle scope \rangle$ is the general database object the operation works on, and Θ is the set of parameters the SMO requires. Figure 3 gives a systematic overview of all SMOs in CoDEL. A relational database table is a two-dimensional structure consisting of columns and rows, hence, SMOs can operate on the level of columns, of rows, or of whole tables. On all three levels there are five basic operations: ADD, DEL, SPLIT, UNITE, and REN. We now introduce the meaningful operations, as shown in Fig. 3. First, CoDEL has two basic operations to create (ADD_{table}) and drop (DEL_{table}) tables as a whole, similar to their counterparts in a standard DDL. Second, CoDEL has a set of operations to modify a table. Hence, CoDEL offers eight table modification SMOs $\langle smo \rangle_{\langle scope \rangle}$ with $\langle scope \rangle \in \{column, row\}$ and $\langle smo \rangle \in \{\text{ADD}, \text{DEL}, \text{SPLIT}, \text{UNITE}\}$. For instance, DEL_{column} removes a column from a given table and SPLIT_{row} partitions a table horizontally, while SPLIT_{column} partitions it vertically. CoDEL defines no SPLIT or UNITE of whole tables, since these operations are restricted to either column or row scope. Third, CoDEL includes two SMOs to rename a table (REN_{table}) and a column (REN_{column}). The renaming of rows is undefined.

Regarding relational completeness, REN_{column}, REN_{table}, DEL_{column}, and DEL_{row} are not necessary, as they are subsumed by the remaining SMOs. However, they are very common (Curino et al. 2008) and included in CoDEL for usability's sake. To summarize, CoDEL is the database evolution language $\mathcal{L}_C$ with:

$$\mathcal{L}_C = \left\{ \begin{array}{llll} \text{ADD}_{table}, & \text{DEL}_{table}, & & \\ \text{ADD}_{column}, & \text{DEL}_{column}, & \text{SPLIT}_{column}, & \text{UNITE}_{column}, \\ \text{ADD}_{row}, & \text{DEL}_{row}, & \text{SPLIT}_{row}, & \text{UNITE}_{row}, \\ \text{REN}_{table}, & \text{REN}_{column} & & \end{array} \right\}$$

All CoDEL SMOs require a set Θ of parameters. Let $inst(o, D)$ be the set of instances of the SMO o with a valid parameterization regarding the database D. For instance, the only parameter to remove a table with $\text{DEL}_{table}(\Theta)$ is the name of an existing table, so $inst(\text{DEL}_{table}(\Theta), D) = \{\text{DEL}_{table}(R) | R \in D\}$. Further, let $inst(\mathcal{L}, D) = \bigcup_{o \in \mathcal{L}} inst(o, D)$ be the set of all validly parameterized SMO instances of the database evolution language $\mathcal{L}$. Then, a CoDEL evolution script s for a database D is a sequence of instantiated SMOs with $s \in inst(\mathcal{L}_C, D_i)^+$, where D_i is the database after applying the i-th SMO.

In the following, we specify the semantics of all CoDEL SMOs. Table 1 summarizes the definition of the semantics based on $\mathcal{L}_{\text{min}}$. The table also shows the SQL-like syntax we propose for the implementation of CoDEL. In the remainder, $R.C = \{c_1, \ldots, c_n\}$ denotes the set of columns of table R and R_i specifies the revision i of the table R. Whenever an SMO does not change the table's name but its columns or rows, we increment this revision counter i to avoid naming conflicts. CoDEL SMOs take tables as input and return tables. According to the SQL standard, tables are multisets. Our $\mathcal{L}_{\text{min}}$ semantics is based on the relational algebra, so tables are sets. However, relational database systems internally manage row identifiers, which are at least unique per table. At the level of SMO implementation, we consider the row identifiers as part of the tables, hence tables are sets. The corresponding multiset semantics of the SMOs can be achieved, by adding a multiset projection of the resulting tables that removes the row identifiers without eliminating duplicates.

ADD$_{table}$ and DEL$_{table}$: The SMOs ADD$_{table}$ and DEL$_{table}$ are the simplified version of their $\mathcal{L}_{\min}$ counterparts. ADD$_{table}(R, \{c_1, \ldots, c_n\})$ requires two parameters, a table name R and a set of column definitions c_i. It creates an empty table with the specified name and schema. DEL$_{table}(R)$ takes only a single parameter, the name of the table to be dropped.

ADD$_{column}$ and DEL$_{column}$: The SMO ADD$_{column}$ adds a new column to an existing table. As parameter ADD$_{column}(R_i, c, f(c_1, \ldots, c_n))$ takes the name R_i of the table, the column definition c of the new column, and a function f. The resulting table is R_{i+1}. ADD$_{column}$ applies the function f to each row in R_i to calculate the row's value for the new column c. The function f is a standard SQL function that can access the existing values of the row, but must not contain nested queries.

DEL$_{column}$ removes a column from a table. Specifically, DEL$_{column}(R_i, c)$ takes the name R_i of an existing table and the name $c \in R_i.C$ of the column that should be removed from R_i. The resulting table is R_{i+1}.

SPLIT$_{column}$ and UNITE$_{column}$: The SMO SPLIT$_{column}$ partitions a table vertically. SPLIT$_{column}$ has a generalized semantics, where the resulting partitioning is allowed to be incomplete and overlapping. The partitioning SMO SPLIT$_{column}(R, (S, \{s_1, \ldots, s_n\}), (T, \{t_1, \ldots, t_m\}))$ takes the name R of the original table, a pair consisting out of a table name S and a set of column names s_i as specification of the first partition and optionally a second pair $(T, \{t_1, \ldots, t_m\})$ as specification of the second partition. The two sets of column definitions are independent. In case $S.C \cap T.C \neq \emptyset$, the columns $S.C \cap T.C$ are copied. In case $S.C \cup T.C \subset R.C$, the partitioning is incomplete. If the second partition is not specified, T is not created. CODEL prohibits empty column sets for S and T, since tables must have at least one column.

UNITE$_{column}$ is the inverse operation of SPLIT$_{column}$ and joins two tables based on a given condition. As parameters, UNITE$_{column}(R, S, T, cond, o)$ takes the names R and S of the original tables, the name T of the resulting table, a join condition $cond$, and the optional boolean o to indicate an outer join. The join condition is an arbitrary SQL predicate without further nesting, so N:M mappings are explicitly allowed. In case $o = \top$, UNITE$_{column}$ performs an outer join, so that no rows from the original tables are lost. In case $o = \bot$ (or not specified) UNITE$_{column}$ performs an inner join. With the inner join, UNITE$_{column}$ loses all rows from R and S that do not find a join partner, since R and S are dropped after the join. More complex join types like e.g. left, right, or full outer joins can be simulated in sequences with other SMOs (Ullman 1988). Note that restricting the join to foreign key relations as other database evolution languages do, does not prevent this information loss. A foreign key does not guarantee that every row in the referenced table is actually referenced by at least one row in the referencing table.

ADD$_{row}$ and DEL$_{row}$: ADD$_{row}$ creates new rows. The values of these new rows are either constants or determined by aggregations functions over the currently exiting rows. ADD$_{row}\big(R_i, G, \{(a_j, f_j(G, V))|1 \leq j \leq m\}, S\big)$ requires the following parameters: the name R_i of the original table, the set of grouping columns $G = \{g_1, \ldots, g_n\} \subseteq R_i.C$, a set of pairs of column name a_j and function f_j, and optionally a new table name S. ADD$_{row}$ produces new rows by grouping table R_i by all columns $g_k \in G$ and calculating the values for the columns a_j with the functions f_j. The functions f_j have to return one value that is either a constant or computed by the standard aggregate functions upon the remaining columns $V = R_i.C \setminus G$. In general, the set of grouping columns is also allowed to be empty resulting in one group and hence, one new row. In the simplest case, e.g.

$$\text{ADD}_{row}\big(Task, \emptyset, \{(task, 'DinnerwithFamily'), (prio, 1), (author, 5)\}, \emptyset\big)$$

merely adds a new task in the *TasKy* version of our example in Fig. 2. ADD$_{row}$ appends new rows to R_i to form its new revision R_{i+1} if the table name S is not given. In this case,

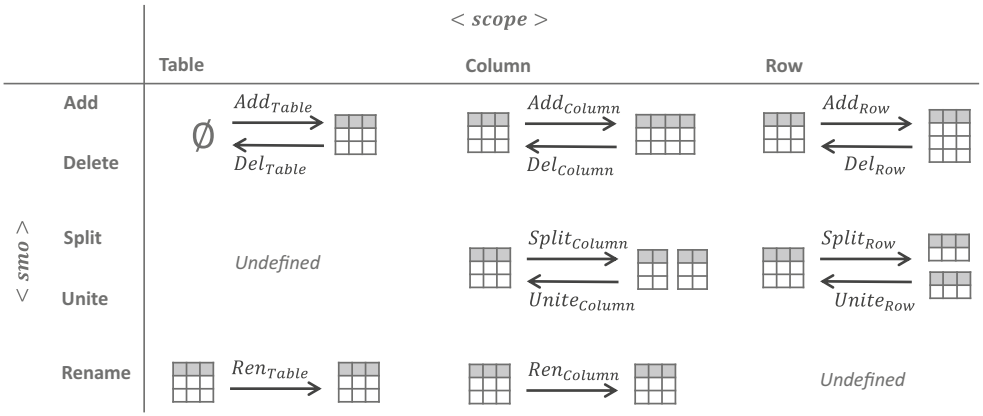


Fig. 3 Structuring of CODEL

we require the column definitions of the new rows to match the original table R_i, hence $\{g_1, \ldots, g_n\} \cup \{a_1, \ldots, a_m\} = R.C$. If the new table name S is specified, ADD_{row} creates S with the newly produced rows and R_i remains available, which is particularly necessary, when the newly created rows have a different set of columns than $R_i.C$. For instance

$$\text{ADD}_{row}(Task, (author), \{(avg\_prio, AVG(prio))\}, Avg\_prio)$$

creates a new table Avg_prio with two columns: the author and the average priority of the respective author's tasks.

DEL_{row} removes rows from a given table. $\text{DEL}_{row}(R_i, cond)$ takes the name of an existing table R_i and a condition $cond$. It removes all rows, which satisfy the condition and evolves the table to R_{i+1}.

SPLIT_{row} and UNITE_{row}: SPLIT_{row} partitions a table horizontally. However, its semantics is more general than standard horizontal partitioning (Ceri et al. 1982). The SMO creates at most two partitions out of a given table—with the partitioning allowed to be incomplete and overlapping—and removes the original table. More precisely, $\text{SPLIT}_{row}(R, (S, cond_S), (T, cond_T))$ takes the name R of the original table, a pair of table name S and condition $cond_S$ as specification of the first partition and optionally a second pair $(T, cond_T)$ as specification of the second partition. Both conditions $cond_S$ and $cond_T$ are independent. If the original tables contain rows that fulfill neither of the conditions, the resulting partitioning is incomplete. Rows that fulfill both conditions are copied resulting in overlapping partitions. In case both conditions hold for

Table 1 Syntax and Semantic of CoDEL operations

SMO:	$\text{ADD}_{table}(R, \{c_1, \ldots, c_n\})$	$\text{DEL}_{table}(R)$
Semantic:	$\text{ADD}(R, \pi_{c_1,\ldots,c_n}(\emptyset))$;	$\text{DEL}(R)$;
Syntax:	`CREATE TABLE` R `(`c_1`,...,`c_n`)`	`DROP TABLE` R
SMO:	$\text{ADD}_{column}(R_i, c, f(c_1, \ldots, c_n))$	
Semantic:	$\text{ADD}\big(R_{i+1}, \pi_{R_i.C \cup \{c \leftarrow f(c_1,\ldots,c_n)\}}(R_i)\big)$; $\text{DEL}(R_i)$;	
Syntax:	`ADD COLUMN` c `AS` f`(`c_1`,...,`c_n`)` `INTO` R_i	
SMO:	$\text{DEL}_{column}(R_i, c)$	
Semantic:	$\text{ADD}\big(R_{i+1}, \pi_{R_i.C \setminus \{c\}}(R_i)\big)$; $\text{DEL}(R_i)$;	
Syntax:	`DROP COLUMN` c `FROM` R_i	
SMO:	$\text{SPLIT}_{column}(R, (S, \{s_1, \ldots, s_n\}), (T, \{t_1, \ldots, t_m\}))$	
Semantic:	$\text{ADD}(S, \pi_{s_1,\ldots,s_n}(R))$; $[\text{ADD}(T, \pi_{t_1,\ldots,t_m}(R))]$; $\text{DEL}(R)$;	
Syntax:	`DECOMPOSE TABLE` R `INTO` S `(`s_1`,...,`s_n`) [,` T `(`t_1`,...,`t_m`)]`	
SMO:	$\text{UNITE}_{column}(R, S, T, cond, o)$	
Semantic:	$o = \bot$: $\text{ADD}(T, R \bowtie_{cond} S)$; $o = \top$: $\text{ADD}(T, R \,\overline{\bowtie}_{cond} S)$; $\text{DEL}(R)$; $\text{DEL}(S)$;	
Syntax:	`[OUTER] JOIN TABLE` R, S `INTO` T `WHERE` $cond$	
SMO:	$\text{ADD}_{row}(R_i, G, \{(a_1, f_1(G,V)), \ldots, (a_m, f_m(G,V))\}, S)$	
Semantic:	S given: $\text{ADD}\Big(S, \gamma_{G, \{f_j(G,V) \to a_j \mid 1 \le j \le m\}}(R_i)\Big)$	
	S not given: $\text{ADD}\Big(R_{i+1}, R_i \cup \gamma_{G, \{f_j(G,V) \to a_j \mid 1 \le j \le m\}}(R_i)\Big)$; $\text{DEL}(R_i)$;	
Syntax:	`AGGREGATE TABLE` R_i `(`g_1`,...,`g_n`)` `WITH` $a_1 = f_1(G,V)$`,...` `[INTO` S`]`	
SMO:	$\text{DEL}_{row}(R_i, cond)$	
Semantic:	$\text{ADD}(R_{i+1}, \sigma_{\neg cond}(R_i))$; $\text{DEL}(R_i)$;	
Syntax:	`REMOVE FROM TABLE` R_i `WHERE` $cond$	
SMO:	$\text{SPLIT}_{row}(R, (S, cond_S), (T, cond_T))$	
Semantic:	$\text{ADD}\big(S, \sigma_{cond_S}(R)\big)$; $\big[\text{ADD}\big(T, \sigma_{cond_T}(R)\big)\big]$; $\text{DEL}(R)$;	
Syntax:	`PARTITION TABLE` R `INTO` S `WITH` $cond_S$ `[,` T `WITH` $cond_T$`]`	
SMO:	$\text{UNITE}_{row}(R, S, T)$	
Semantic:	$\text{ADD}\big(T, \pi_{R.C \cup \{\omega \to a_i \mid a_i \in S.C \setminus R.C\}}(R) \cup \pi_{S.C \cup \{\omega \to a_i \mid a_i \in R.C \setminus S.C\}}(S)\big)$; $\text{DEL}(R)$; $\text{DEL}(S)$;	
Syntax:	`MERGE TABLE` R, S `INTO` T	
SMO:	$\text{REN}_{table}(R, R')$	$\text{REN}_{column}(R_i, c, c')$
Semantic:	$\text{ADD}(R', R)$; $\text{DEL}(R)$;	$\text{ADD}\big(R_{i+1}, \rho_{c'/c}(R_i)\big)$; $\text{DEL}(R_i)$;
Syntax:	`RENAME TABLE` R `INTO` R'	`RENAME COLUMN` c `IN` R_i `TO` c'

all rows, i.e., $cond_S = \top$ and $cond_T = \top$, both S and T are complete copies of R. Hence, SPLIT_{row} subsumes the functionality of a copy operation that can be found in other database evolution languages. If $cond_T$ is not specified, SPLIT_{row} does not create table T.

UNITE_{row} is the inverse operation of SPLIT_{row}; it merges two given tables along the row dimension and removes the original tables. As parameters $\text{UNITE}_{row}(R, S, T)$ requires, the names R and S of the original tables and the name T of the resulting table. The schema of R and S are not required to by equivalent. In case both schemas differ, T contains null values (ω) in the respective cells. UNITE_{row} eliminates duplicates in T. In case R and S contain equivalent rows, these rows will show up only once in T.

REN_{table} and REN_{column}: The last two SMOs rename schema elements. $\text{REN}_{table}(R, R')$ renames the table with the name R into R'. $\text{REN}_{column}(R_i, c, c')$ renames the column c in table R_i into c', which results in table R_{i+1}.

Now, we have a precise definition of both the syntax and semantics of CoDEL; summarized in Table 1. In the next section, we show that the introduced SMOs are powerful enough to cover common evolution scenarios and provide a formal guarantee for relational completeness as well.

3 Completeness Properties

Using a comprehensive database evolution language like CoDEL, greatly simplifies a database developer's life, since it serves as an intuitive documentation, prevents mistakes, and facilitates to automate further tasks that come along with the database evolution. In order to benefit from these advantages, there is one fundamental requirement: the database evolution language must be complete. A complete database evolution language allows the developer to intuitively specify any intended evolution with the given SMOs. In contrast, whenever the developer has to fall back on traditional SQL, all the formerly solved problems return. We consider two levels of completeness:

- **Practical Completeness:** Given the evolution history of existing projects, e.g. the Wikimedia project, a database evolution language must allow to model the same evolution exclusively with the given SMOs.
- **Relational Completeness:** Relational completeness is a formal property. It guarantees that any evolution that can be expressed with relational algebra expressions can be expressed with the given SMOs as well.

We show CoDEL's practical completeness in Section 3.1, formally validate its relational completeness in Section 3.2, and therefore confidently consider CoDEL as a feasible database evolution language for any evolution scenario.

3.1 Practical Completeness

We use the database evolution benchmark from Carlo Curino et al. (Curino et al. 2008) that is based on actual evolution histories of open source projects. They modeled the evolution history of Wikimedia (171 versions) with their database evolution language PRISM (Curino et al. 2012) and we did the same with CoDEL. CoDEL proved to be capable of providing the database schema in each version exactly according to the benchmark and migrate the data in a meaningful way. So, CoDEL is practical complete and feasible to handle such real world scenarios.

Figure 4 summarizes characteristics of the 209 SMOs long CoDEL evolution history. Particularly, Fig. 4a shows how often each SMO has been used in total. We account the dominance of simple SMOs like adding and removing both columns and tables mainly to the restricted database evolution support current DBMSes provide. Still, there are more complex evolutions requiring the other SMOs as well, so there is a need for more sophisticated database evolution support. Figure 4b shows the number of SMOs per evolution step from one version to its successor. Again, we see the same pattern: there are mostly simple evolutions with only a few SMOs, but also some more complex ones. In general, the characteristics of the evolution with CoDEL is obviously very similar to the same evolution modeled with PRISM (Curino et al. 2008), however there are slight differences, since the respective sets of SMOs differ. For instance the complex evolution from version v06696 to v06710 takes 31 SMOs instead of 92 SMOs due CoDEL's powerful decompose SMO. The sequence of CoDEL SMOs, used to model Wikimedia's 171 schema versions, is accessible online.[1]

3.2 Relational Completeness

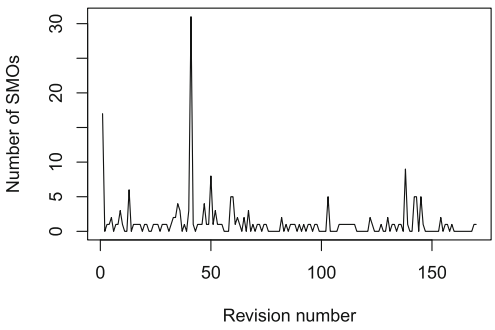
To show the relational completeness of CoDEL, we argue that it is at least as powerful as $\mathcal{L}_{\text{min}}$ (Section 2), which is relationally complete by definition. There is always a semantically equivalent expression in CoDEL for any expression in $\mathcal{L}_{\text{min}}$. The $\text{DEL}(R)$ operation from $\mathcal{L}_{\text{min}}$ is trivial, since it is equivalent to CoDEL's $\text{DEL}_{table}(R)$. On the contrary, $\text{ADD}(R, \epsilon)$ from $\mathcal{L}_{\text{min}}$ is more complex, as ϵ covers the power of the relational algebra. Since both the relational algebra and CoDEL are closed languages, it is reasonable to address each operation of the relational algebra separately. We show that, for each operation from the relational algebra, there is a semantically equivalent sequence of SMOs in CoDEL.

[1] https://wwwdb.inf.tu-dresden.de/research-projects/projects/inverda.

Fig. 4 Wikimedia Benchmark: The history of schema versions modeled with CoDEL

SMO	#occurences
CREATE TABLE	42
DROP TABLE	10
RENAME TABLE	1
ADD COLUMN	93
DROP COLUMN	20
RENAME COLUMN	37
JOIN	0
DECOMPOSE	4
MERGE	2
PARTITION	0

(a) SMO usage.



(b) SMOs per Revision.

We assume the basic relational algebra (Codd 1970) and add common extensions like the extended projection, aggregation, and outer joins. For the basic relational algebra we consider selection, renaming, projection, cross product, as well as union and difference of sets—the minimal set of operations that covers the whole relational algebra including other operations like intersection and division (Ullman 1988). We intentionally exclude other extensions like the transitive closure and sorting, since CoDEL is non-recursive and set-based. We maintain these characteristics, since they proved to be a reasonable trade-off between expressiveness and usability, however, they are open for further research. With respect to implementations based on current database management systems, the distinction between different types of null values (Zaniolo 1984) is not considered. For instance UNITE_{row} adds null values in columns, which existed in only one input table, losing the information, whether a value was null before or did not exist at all. The following sections consider the relational algebra operations (Ullman 1988) plus the chosen extensions and show that CoDEL is capable to obtain the semantically equivalent results.

Relation: R The basic elements of the relational algebra are relations. They contain the data and are directly accessible by CoDEL as tables. Whenever one table is required multiple times within a relational algebra expression, CoDEL allows to copy them using $\text{SPLIT}_{row}(R, (S, \top), (T, \top))$.

Selection: $\sigma_{cond}(R)$ The relational selection operation returns the subset of rows from R, where each row satisfies the condition $cond$. CoDEL's $\text{SPLIT}_{row}(R, (S, cond))$ is semantically equivalent, which directly follows from the semantics definition in Table 1.

Rename: $\rho_{c'/c}(R_i)$ Renaming a column is subsumed by the extended projections, however, we include it here for completeness. CoDEL's obvious semantic equivalent according to Table 1 is $\text{REN}_{column}(R_i, c, c')$.

Extended Projection: $\pi_P(R)$ We will consider the extended projection, as it subsumes the traditional projection as well. The extended projection defines a new set of columns, whose values are computed by functions depending on the existing columns. The projection $P = \{f_k(R.C) \to a_k | 1 \leq k \leq m\}$ produces a relation with m columns, each being computed by a function $f_k(R.C)$ taking the $n = |R.C|$ columns from R as input. For instance, we project the task table in version *TasKy* in Fig. 2 to $\pi_{task \to task, (prio==1) \to isUrgent}(Task)$, so we return the task and a boolean stating whether its priority is 1 or not. Algorithm 1 describes the CoDEL sequence for the relational projection operation in general.

Algorithm 1 CoDEL sequence for relational projection

```
1: for k = [1..m] do
2:     ADD_column(R_{i+k-1}, a'_k, f_k(r_1, ..., r_n));
3: for r_j ∈ R.C do
4:     DEL_column(R_{i+m+j-1}, r_j);
5: for k = [1..m] do
6:     REN_column(R_{i+m+n+k-1}, a'_k, a_k);
```

Without loss of generality, we use for-loops to iterate over the attribute sets. Since this is only schema depending and data independent, it does not extend the expressiveness of the database evolution language but is simply a short notation. The first SMO adds a new column, with a masked name, for each column of the output table. This allows to compute the new values based on all existing ones. Afterwards, we drop the old columns, rename the new columns to their unmasked name, and remove all intermediate tables. In our example above, we first add the two columns $task'$ and $isUrgent'$ and computer the projected values. Afterwards, we remove the original columns *name*, *prio*, and *author* and finally rename the projected columns to *task* and *isUrgent*. Applying the semantics definitions

of the CoDEL SMOs to Algorithm 1 results in the desired extended projection, as we will show now. The concrete line of the CoDEL sequence in Algorithm 1, which is applied in the semantics computation, is indicated by the numbers above the equal signs.

$$R_{i+1} \overset{l.2}{=} \pi_{r_1,\ldots,r_n,f_1(r_1,\ldots,r_n)\to a'_1}(R_i) \quad (1)$$

$$R_{i+m} \overset{l.1,2}{=} \pi_{r_1,\ldots,r_n,f_1(r_1,\ldots,r_n)\to a'_1,\ldots,f_m(r_1,\ldots,r_n)\to a'_m}(R_i) \quad (2)$$

$$R_{i+m+1} \overset{l.4}{=} \pi_{r_2,\ldots,r_n,a'_1,\ldots,a'_m}(R_{i+m}) \quad (3)$$

$$\begin{aligned} R_{i+m+n} &\overset{l.3,4}{=} \pi_{a'_1,\ldots,a'_m}(R_{i+m}) \\ &= \pi_{f_1(r_1,\ldots,r_n)\to a'_1,\ldots,f_m(r_1,\ldots,r_n)\to a'_m}(R_i) \end{aligned} \quad (4)$$

$$R_{i+m+n+1} \overset{l.6}{=} \pi_{a'_1\to a_1,a'_2,\ldots,a'_m}(R_{i+m+n}) \quad (5)$$

$$\begin{aligned} R_{i+m+n+m} &\overset{l.5,6}{=} \pi_{a'_1\to a_1,\ldots,a'_m\to a_m}(R_{i+m+n}) \\ &= \underline{\underline{\pi_{f_1(R_i.C)\to a_1,\ldots,f_m(R_i.C)\to a_m}(R_i)}} \end{aligned} \quad (6)$$

In Eq. 1, we apply Line 2 from Algorithm 1 for the first projected column a'_1, which merely adds the calculated value and the column name to the projection clause. According to the loop on Lines 1 and 2 we do this for all projected columns as shown in Eq. 2. In the second loop on Lines 3 and 4, we drop the original columns of the table, since they are no longer required for the calculation. Equation 3 shows the result after dropping the first column, while Eq. 4 shows the table after full iteration of the second loop. Finally, we replace the masked names with the originally intended names of the projected columns on Lines 5 and 6 to first obtain Eq. 5 and finally the extended projection in Eq. 6.

Outer Join: $R \mathbin{⟗_p} S$ The outer join is another common extension to the traditional relational algebra. Beyond the rows according to an inner join, it also includes those rows in the result, which did not find a join partner. The missing values for columns of the other table are filled with null values. Obviously, CoDEL's $\textsc{Unite}_{column}(R, S, T, p, \top)$ is semantically equivalent, since we explicitly introduced the option to perform outer joins.

Cross Product: $R \times S$ The cross product produces a row in the output table for each pair of rows from the input tables. Algorithm 2 describes the CoDEL sequence to obtain the relational cross product. We add a new column j to both tables with $j \notin R_i.C$ and $j \notin S_k.C$. The default value of j is a constant 1, so we can perform an inner join on j, such that there will be one row in the output table for each pair of rows from the two input tables. Finally, we remove the additional column j and summarize the outcome to the cross product of R and S. Thereby, we show the semantic equivalence between the relational cross product and the presented sequence of CoDEL SMOs.

Algorithm 2 CoDEL sequence for relational cross product

```
1: ADD_column(R_i, j, 1);
2: ADD_column(S_k, j, 1);
3: UNITE_column(R_{i+1}, S_{k+1}, T_0, R_{i+1}.j = S_{k+1}.j, ⊥);
4: DEL_column(T_0, j);
```

$$R_{i+1} \overset{l.1}{=} \pi_{r_1,\ldots,r_n,1\to j}(R_i) = \{(r_1,\ldots,r_n,1) | (r_1,\ldots,r_n) \in R_i\} \quad (7)$$

$$S_{k+1} \overset{l.2}{=} \{(s_1,\ldots,s_m,1) | (s_1,\ldots,s_m) \in S_k\} \quad (8)$$

$$\begin{aligned} T_0 &\overset{l.3}{=} R_{i+1} \bowtie_{R_{i+1}.j=S_{k+1}.j} S_{k+1} \\ &= \{(r_1,\ldots,r_n,s_1,\ldots,s_m,1) | (r_1,\ldots,r_n) \\ &\quad \in R_i, (s_1,\ldots,s_m) \in S_k\} \end{aligned} \quad (9)$$

$$\begin{aligned} T_1 &\overset{l.4}{=} \{(r_1,\ldots,r_n,s_1,\ldots,s_m) | (r_1,\ldots,r_n) \in R_i, (s_1,\ldots,s_m) \in S_k\} \\ &= \underline{\underline{R \times S}} \end{aligned} \quad (10)$$

Aggregate: $\gamma_{G,F}(R)$: The aggregation is a typical extension to the relational algebra. The rows are grouped by one set of columns $G = \{g_1,\ldots,g_n\} \subseteq R.C$. Additional columns $A = \{a_i | 1 \le i \le p\}$ are computed by functions $F = \{f_i(G, V) \to a_i | a_i \in A\}$ with $V = \{v_1,\ldots,v_m\} = R.C \setminus G$. These functions may contain values from grouping columns G, aggregate functions on the remaining columns in V, constants, and arithmetic functions. CoDEL contains a dedicated operation $\textsc{Add}_{row}(R, G, F, S)$. It writes the result of the aggregation to the new table S. According to the semantics definition in Table 1, the semantics of Algorithm 3 equals the discussed aggregation semantics from the relational algebra.

Algorithm 3 CoDEL sequence for relational aggregation operation

```
1: AGGREGATE TABLE R INTO T WITH G, F;
2: DROP TABLE R;
```

$$T \overset{l.1}{=} \gamma_{(g_1,\ldots,g_n),\{f_i\to a_i | 1\le i\le o\}}(R) = \underline{\underline{\gamma_{G,F}(R)}} \quad (11)$$

Union: $R \cup S$ The relational union operation merges the rows from both input tables to the one output table including an elimination of duplicates. Using the SMO $\textsc{Unite}_{row}$, CoDEL provides a semantic equivalent to the relational union operation.

Algorithm 4 CoDEL sequence for relational union operation

```
1: UNITE_row(R, S, T);
```

$$T \overset{l.1}{=} \pi_{R.C}(R) \cup \pi_{S.C}(S) = \underline{\underline{R \cup S}} \quad (12)$$

Please note, that the union in the relational algebra requires R and S to have identical sets of attributes ($R.C = S.C$), which justifies the simplification step.

Difference: $R \setminus S$ The relational difference returns all rows, which occur in the first, but not in the second table. Analogous to the union, it requires R and S to have identical sets of columns ($R.C = S.C$). Algorithm5 describes the CoDEL sequence to obtain the relational difference, where ω denotes a null value. We add a new column j to S_k with $j \notin S_k.C$ and the default value 1. The outer join on all column pairs in $\{(R_i.c_i, S_k.c_i) | c_i \in R_i.C\}$ (remember $R_i.C = S_k.C$), results in a table containing all rows which were in at least one of the two input tables. However, all rows that occurred in S_k have the value 1 in the column j and are removed by the third SMO. All rows which occurred exclusively in R have a null value ω in the column j and remain as a result. Applying the semantics definition of the SMOs finally leads to the relational difference operation. Please note, that $(r_1, \ldots, r_n) \notin S_k$ is equal to $(r_1, \ldots, r_n, 1) \notin S_{k+1}$ due to the first step.

Algorithm 5 CoDEL sequence for relational difference operation

1: $\text{ADD}_{column}(S_k, j, 1)$;
2: $\text{UNITE}_{column}(R_i, S_{k+1}, T_0, (R_i.c_1 = S_{k+1}.c_1 \wedge \ldots \wedge R_i.c_n = S_{k+1}.c_n), \top)$;
3: $\text{DEL}_{row}(T_0, j \neq \omega)$;
4: $\text{DEL}_{column}(T_1, j)$;

$$S_{k+1} \overset{l.1}{=} \pi_{s_1,\ldots,s_m,1 \to j}(S_k) \tag{13}$$

$$\begin{aligned} T_0 \overset{l.2}{=} & R_i \rtimes S_{k+1} \\ = & \{(r_1, \ldots, r_n, 1) \mid (r_1, \ldots, r_n) \in R_i, (r_1, \ldots, r_n, 1) \in S_{k+1}\} \\ & \cup \{(r_1, \ldots, r_n, 1) \mid (r_1, \ldots, r_n) \notin R_i, (r_1, \ldots, r_n, 1) \in S_{k+1}\} \\ & \cup \{(r_1, \ldots, r_n, \omega) \mid (r_1, \ldots, r_n) \in R_i, (r_1, \ldots, r_n, 1) \notin S_{k+1}\} \end{aligned} \tag{14}$$

$$\begin{aligned} T_1 \overset{l.3}{=} & \sigma_{\neg(j \neq \omega)}(T_0) = \sigma_{(j=\omega)}(T_0) \\ = & \{(r_1, \ldots, r_n, \omega) | (r_1, \ldots, r_n) \in R_i, (r_1, \ldots, r_n, 1) \notin S_{k+1}\} \end{aligned} \tag{15}$$

$$\begin{aligned} T_2 \overset{l.4}{=} & \pi_{R.C}(T_1) \\ = & \{(r_1, \ldots, r_n) | (r_1, \ldots, r_n) \in R_i, (r_1, \ldots, r_n) \notin S_k\} = \underline{\underline{R_i \setminus S_k}} \end{aligned} \tag{16}$$

In Eq. 13, we apply Line 1 from Algorithm 5 so that S_{k+1} contains the added column j with the value 1. Equation 14 shows the joint table T_0 according to the outer equi join on all columns except j, such that T_0 is basically a duplicate-eliminating union of R and S, where any tuple occurring is S has $j = 1$ (Line 2). In Eq. 15, we use this and filter those tuples in S out (Line 3), so we can finally drop the auxiliary column j in Eq. 16 (Line 4) to obtain the desired relational difference operation.

Finally, we successfully showed that CoDEL provides a semantic equivalent for each relational algebra expression, which makes it equally expressive as $\mathcal{L}_{\text{min}}$. Hence, CoDEL is a relational complete database evolution language and a sound foundation for further research.

4 Semi-automatic Variant Co-evolution

CoDEL is a powerful database evolution language that couples the evolution of both the schema and the data in simple SMOs and thereby captures the developers' intention. This facilitates valuable simplifications and supporting tools for database developers. We present semi-automated variant co-evolution as an example. While SQL hides the developer's intent between the lines, namely between DDL and DML and DQL statements, CoDEL represents the intent explicitly, which essentially allows us to consolidate the intentions of multiple evolutions in the first place.

Figure 5 illustrates variant co-evolution using the example from Fig. 2: Given the core application *TasKy*, we derive a variant *Do!* with a sequence of CoDEL SMOs. When evolving the core *TasKy* to the next version *TasKy2*, we also want to co-evolve the variant. Intuitively, the new variant, called *DoTwo!*, should consolidate the intended changes from both the evolution to *Do!* and the evolution to *TasKy2*. Generally speaking, given a sequence of SMOs S_V that evolves the core database D^{Core} to a variant $\text{D}^{Variant}$, and another sequence of SMOs $S_{C'}$ that evolves the core database D^{Core} to a new version $D^{Core'}$, the goal is to propose a co-evolved variant $\text{D}^{Variant'}$ that is based on the evolved core $\text{D}^{Core'}$ and preferably maintains the intents of S_V in $S_{V'}$. So, we **consolidate the intentions of $\mathbf{S_{V'}}$ with the intention of $\mathbf{S_{C'}}$**. To summarize the problem of variant co-evolution:

$$\begin{aligned} Given: \quad & D^{Core} \xrightarrow{S_V} D^{Variant} \text{ and } D^{Core} \xrightarrow{S_{C'}} D^{Core'} \\ Wanted: \quad & D^{Core'} \xrightarrow{S_{V'}} D^{Variant'} \end{aligned}$$

We present VaCo, a tool utilizing the strength of CoDEL to semi-automate the variant co-evolution and thereby release developers from the repetitive task of applying core-evolutions to the respective variants manually. To resolve contradictions or ambiguities of different intents, we chose a semi-automatic approach to keep the developer in the loop and clarify such conflicts by answering very simple questions. VaCo helps avoiding faulty evolutions and allows the developer to focus on the actual implementation task. By merely combining the SMOs of both the variant evolution S_V and the core evolution $S_{C'}$, VaCo can propose a co-evolved variant evolution $S_{V'}$ to the developer.

VACO follows a two level algorithm: globally consolidating sequences of SMOs by locally consolidating pairs of SMOs. So, on the local level, VACO combines one SMO of the variant with one SMO of the core evolution. This combination process is backed by a *consolidation matrix* that defines the intuitively expected result for any pair of SMOs. We present the SMO consolidation including the interface of the consolidation matrix in Section 4.1. This consolidation matrix is exchangeable, however, we propose one specific consolidation matrix for the general case in Section 4.2. For several combinations, the consolidation matrix can define a range of possibly intended results. In these situations, VACO prompts the developer for clarification of his/her intent. On the global level, the algorithm extends the local consolidation of two single SMOs to two sequences of SMOs, which finally allows semi-automatic variant co-evolution as we show in Section 4.3.

4.1 Consolidation of SMOs

Combining two SMOs, one from the core and one from the variant, is the very foundation of VACO. This SMO composition operation $\circ : \{SMO_{core}\} \times \{SMO_{var}\} \rightarrow SMO^*$ takes a core SMO_{core} and a variant SMO_{var}, and returns a sequence SMO^* that can be executed after the SMO_{core}. Executing the derived sequence of SMOs after SMO_{core} should represent the intention of SMO_{var} as close as possible. The result of $\circ$ is a sequence of SMOs since one SMO_{var} may result in multiple SMOs after the consolidation. For instance, the core partitions a table and the variant's evolution has to be applied to both partitions.

The composition operation $\circ$ is not commutative. As shown in Fig. 5, the core evolution remains unchanged. The co-evolved variant SMOs S'_V have to start at the evolved core $Core'$. On the application level, this allows to keep the evolved core application unchanged but requires corresponding evolution of the variant application. Hence, the SMO consolidation $\circ$ is not commutative to ensure that SMO_{Core} remains unchanged and the derived sequence SMO^* is applicable after this SMO_{Core}.

The result of the SMO composition $\circ$ is the intuitively expected combination of the two input SMOs. However, the intuitive expectation is hard to grasp formally. Hence, VACO stores these intuitive expectations explicitly in the consolidation matrix. The consolidation matrix represents the consolidation operation $\circ$. For each possible pair of SMOs the consolidation matrix contains the intuitively expected outcome. VACO can handle any consolidation matrix matching this interface. In this article, we propose one possible consolidation matrix for the general case, but VACO is by no means limited to it. Based on the example in Fig. 5, let us consider e.g. the consolidation of the variant's partitioning with the core's join SMO. Intuitively, the co-evolved variant should also partition the joint table.

$$\text{UNITE}_{column}(Task, Author, Task, Task.author = Author.id, \perp) \circ \text{SPLIT}_{row}(Task, (Todos, prio = 1), (Archive, prio > 1)) = (\text{SPLIT}_{row}(Task, (Todos, prio = 1), (Archive, prio > 1)))$$

In this particular example, the variant SMO remains unchanged, though this is no general rule. Thanks to the well

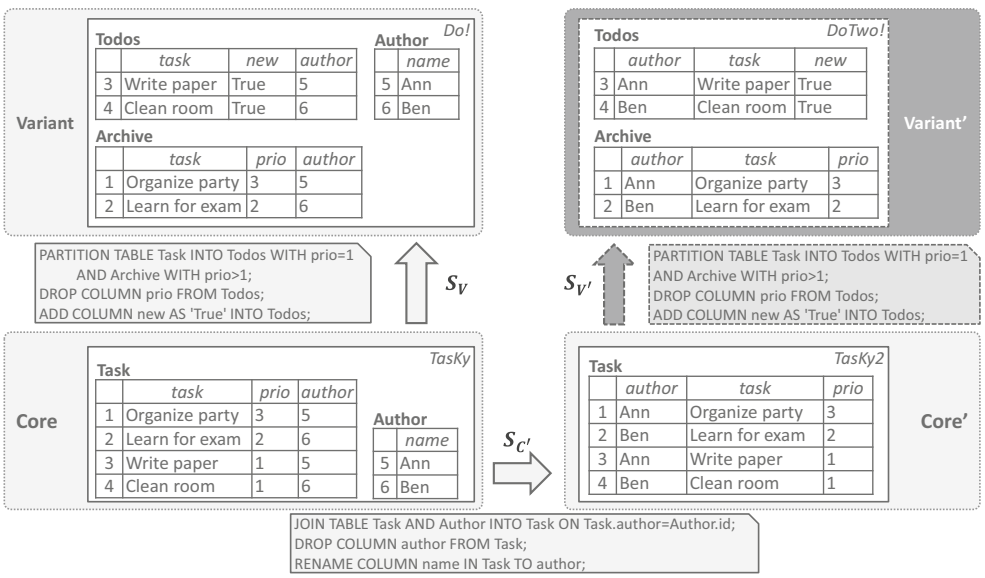


Fig. 5 Variant Co-Evolution with the running example

defined interface of ∘, VACO clearly separates the definition of the intuitive expectations from the effective algorithm.

4.2 General Consolidation Matrix

We propose a consolidation matrix containing each possible combination of SMO_{core} and SMO_{var} and the intuitively expected output for the general case. In specific domains it seems to be reasonable to adapt this matrix accordingly. As SMOs are compact and intuitive, developers can easily adapt the proposed sequence of SMOs and tweak it to the concretely expected variant.

Table 2 shows all combinations of a core SMO and a variant SMO. Most pairs can be consolidated automatically (✓, 108 times). Some combinations of SMOs are contradicting (✗, 18 times), hence the respective SMO_{var} cannot be executed at all. Further, there are three different kinds of user interaction (?, 18 times in total). First, the developer has to resolve naming conflicts (?, 12 times). Second, the developer has to choose one out of two possible paths for a consolidation when the core splits a table (?, 3 times). And finally, the developer has to extend the list of columns of an SMO_{var}=SPLIT_{column} in case there are newly created columns in the core (?, 3 times). All questions are comprehensible and easy to answer by selecting an option or typing a name, which significantly simplifies the developer's effort for the variant co-evolution. In the following, we will explain the consolidation matrix in more detail. First, we discuss how to consolidate SMOs on table level (SMO_{table}) with any other SMO. Afterwards, we go through the consolidation matrix line by line to elaborate on the consolidation of an SMO_{core} both on row (SMO_{row}) and on column (SMO_{column}) level with any other SMO_{var}.

SMO_{table} ∘ $SMO_{table,row,column}$ and **$SMO_{row,column}$ ∘ SMO_{table}** : Consolidating SMOs that work on different input tables is trivial, as they do not interfere at all, so a core's or a variant's ADD_{table} can never conflict with any other SMO except when naming conflicts. Any combination of ADD_{table} SMOs and REN_{table} SMOs might require the developer to assign a new name to the variant table in case it should have the same name as a core table. If the core deletes a table that also serves as input for SMO_{var}, the consolidation will ignore SMO_{var} as the concept was intentionally removed from the database in the core evolution. If the variant removes a table that is used by SMO_{core}, VACO can easily propagate this to the target tables of SMO_{core} and delete those tables as well. For SPLIT_{column} and SPLIT_{row}, we drop both resulting tables. We ignore the variant SMO DEL_{table} if the core SMO is either UNITE_{column} or UNITE_{row} i.a. on the table to drop, as the variant needs to keep the data of the other input table. Given a REN_{table} in the core evolution, we can easily rename the input table in SMO_{var}, however, this does not work in the other direction. Particularly for those SMO_{core} that split or unite tables, we ask the developer to give names for the newly created tables as well.

SMO_{row} ∘ $SMO_{row,column}$: When the core adds or removes a tuple from a table, we propagate all SMO_{var}

Table 2 Compatibility Matrix for SMO Consolidation

	Variant SMO											
Core SMO	ADD_{table}	DEL_{table}	REN_{table}	ADD_{row}	DEL_{row}	SPLIT_{row}	UNITE_{row}	SPLIT_{column}	UNITE_{column}	ADD_{column}	DEL_{column}	REN_{column}
ADD_{table}	$?_N$	✓	$?_N$	✓	✓	✓	✓	✓	✓	✓	✓	✓
DEL_{table}	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
REN_{table}	$?_N$	✓	$?_N$	✓	✓	✓	✓	✓	✓	✓	✓	✓
ADD_{row}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DEL_{row}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SPLIT_{row}	✓	✓	$?_N$	✓	✓	✓	$?_P$	✓	$?_P$	✓	✓	✓
UNITE_{row}	✓	✗	$?_N$	✓	✓	✓	✓	$?_E$	✓	✓	✓	✓
SPLIT_{column}	✓	✓	$?_N$	✗	✗	✓	✓	✗	✓	$?_P$	✓	✓
UNITE_{column}	✓	✗	$?_N$	✓	✓	✓	✓	$?_E$	✓	✓	✓	✓
ADD_{column}	✓	✓	✓	✓	✓	✓	✓	$?_E$	✓	$?_N$	✓	$?_N$
DEL_{column}	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓
REN_{column}	✓	✓	✓	✓	✓	✓	✓	✓	✓	$?_N$	✓	$?_N$

without any change, as neither ADD_{row} nor DEL_{row} change the structure of the respective table. The two tables resulting from a core's SPLIT_{row} have the same structure as the input table, which allows propagating any SMO_{var} simply to both resulting tables as well. Merely when consolidating a core's SPLIT_{row} with a variant SMO that has two input tables (UNITE_{row} and UNITE_{column}), the developer needs to decide to which output of the SMO_{core} to propagate the SMO_{var}. Further, a core's UNITE_{row} produces one output table whose columns are the union of the two input tables. Hence, any SMO_{var} working on one input table does also work on the output table and can be propagated naïvely. The only SMO_{var} to take special care of is SPLIT_{column}. We offer the developer to extend the column sets of the output tables as the input table from the updated core may have more columns, now.

$\text{SMO}_{\text{column}} \circ \text{SMO}_{\text{row,column}}$: Finally, we discuss the core SMOs on column level focusing on the ambiguous combinations that require an interaction of the developer. Given the SMO_{core} extends the set of columns (UNITE_{column} and ADD_{column}), the developer needs to adjust the column set of the variants SPLIT_{column}. Given the SMO_{core} is SPLIT_{column}, it is not possible to also apply a variant's SPLIT_{column} since the original set of column might be incompletely/redundantly distributed over the two output tables. Hence the original concept which should be decomposed in the variant does not exist any longer. However, VaCo can propagate an ADD_{column} through such a SPLIT_{column} by explicitly asking the developer to which output table(s) to propagate the new column. Finally, if both the core and the variant add a new column or rename an existing one, we have to check for naming conflicts and potentially ask the developer for adjustments in the SMO_{var}. In all other cases, there are no conflicts when consolidating the two SMOs.

Summing up, we can consolidate most pairs of SMOs automatically. Obviously there are destructive SMO_{core}, removing either tables or columns, that render certain SMO_{var} impossible. Furthermore, there are a few pairs that require the developer's interaction. The questions VaCo asks the developer are short and simple, keeping the whole semi-automatic co-evolution process convenient. Mainly, we ask the developer to clarify naming conflicts, extend the columns set of SPLIT_{column}, or to choose one output table of splitting SMOs. The proposed consolidation results meet the intuitive expectations and should cover the general case, significantly simplifying the co-evolution.

4.3 Consolidation of Evolutions

Both the variant and the core evolution are defined with sequences of SMOs S_V and $S_{C'}$. To co-evolve the variant and obtain $S_{V'}$, we extend the SMO composition operation $\circ$ from single SMOs to sequences of SMOs. The sequence composition $\bullet : S_V \times S_{C'} \rightarrow SMO^*_{var'}$ returns the sequence of SMOs $S_{V'}$ starting at the new core version. To consolidate whole sequences of SMOs, the general idea is to repeatedly apply the SMO composition $\circ$. Figuratively speaking, VaCo takes each variant SMO from S_V and propagates each one through the core evolution $S_{C'}$ while applying the SMO composition $\circ$.

Figure 6 illustrates this algorithm for the co-evolution process of a variant V created by $C \xrightarrow{S_V} V$ with the core evolution $C \xrightarrow{S_{C'}} C'$. The final result is the co-evolved variant evolution $C' \xrightarrow{S_{V'}} V'$. In each depicted step VaCo propagates one variant SMO from S_V through all core SMOs in $S_{C'}$. For each pair of SMOs, VaCo first checks, whether their sets of input tables overlap. If not, VaCo can simply proceed with the next core SMO. If yes, VaCo applies the composition operation $\circ$ presented before. For the first SMO v_1 in Fig. 6 this works just fine and VaCo appends the resulting SMOs to $S_{V'}$.

Revisiting the example in Fig. 5: the partitioning of the Task table is the first SMO of the variant *Do!*. When propagating it through the join SMO of the core, VaCo simply partitions the Task table resulting from the join. The subsequent dropping and renaming of columns in the core evolution can be consolidated directly with the variant's partitioning SMO. So, VaCo appends the partitioning SMO to the evolved core.

However, already in the second step, we cannot immediately propagate the variant SMO v_2 through the core evolution, since it potentially works on different tables. The first variant SMO v_1 might e.g. rename a table. To compensate for such evolutions, VaCo adapts v_2 to represent the intended evolution on the tables of the initial core schema. This tracing process needs to be inverted after propagating the respective SMO_{var} through the core evolution to obtain the correct $S_{V'}$. VaCo continues until all variant SMOs are propagated through the core evolution and appended to $S_{V'}$.

In our example (Fig. 5), dropping the prio column is the next variant SMO. When tracing this back, VaCo hits the partitioning SMO, which creates new tables, hence breaks the tracing of tables. When VaCo appends this SMO directly to the co-evolved variant SMOs, it drops the column from the new Todos table as expected. The same applies to the final rename column SMO of the variant. We obtain the co-evolved variant as expected and VaCo proposes it to the developer without any further interaction required.

Finally, one remark on the escalation of contradicting SMOs: Whenever VaCo cannot consolidate a variant SMO s, because s contradicts the core evolution (marked with ✗ in Table 2), all subsequent variant SMOs that rely on s can not

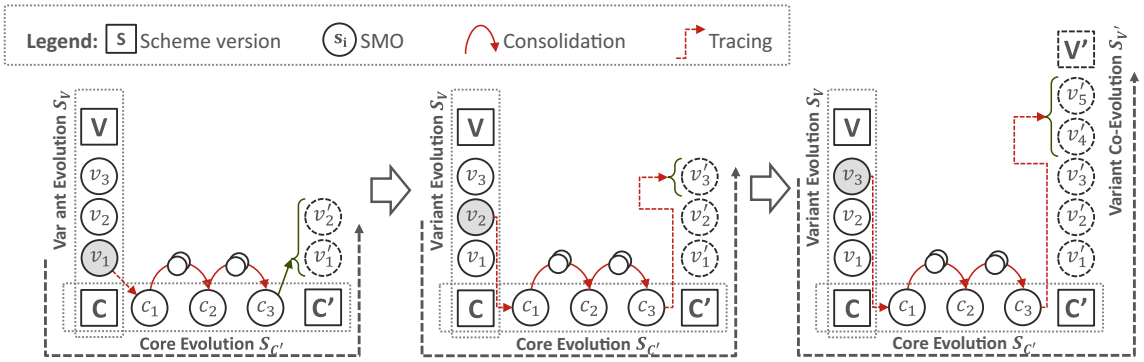


Fig. 6 Consolidation of Sequences of SMOs

be consolidated with the core as well. When e.g. the variant adds a column to a table, however the core evolution intends to drop this table, VACO has to ignore the variant SMO, since the core evolution $S_{C'}$ must not be changed. This also affects all subsequent variant SMOs that rely on the added column in the variant. As the core concepts, on which the variant was based, are removed by the core evolution, the developer has to rethink this part of the variant either way. Pragmatically, VACO simply ignores the variant SMO and leaves it to the developer to integrate the contradicting variant.

Summing up, we use the richer semantics of CODEL to semi-automatically co-evolve a variant with the evolving core. This increases the speed and quality of the development, as VACO supports the developer with the repetitive and error-prone task of co-evolving the variant. So, we showed one useful approach to use CODEL's richer semantics to simplify the database evolution process.

5 Related Work

Database evolution is a well recognized topic in the database research community (Rahm and Bernstein 2006; Terwilliger et al. 2012). There is a number of approaches to increase comfort and efficiency in database evolution, e.g. by defining a schema evolution aware query language (Roddick 1992). Another approach is to define database evolution languages in a graph-based way (Papastefanatos et al. 2008). This allows modeling dependencies between different artifacts in the information system and applying changes globally. Furthermore, MeDEA (Domínguez et al. 2008) provides a general framework to describe database evolution in the context of evolving applications. MoDEF (Terwilliger et al. 2010a) introduces an IDE extension to automate the co-evolution of the evolving client schemas and the store.

Currently, PRISM (Curino et al. 2012) appears to provide the most advanced database evolution tool including an SMO-based database evolution language. PRISM was first introduced in 2008 and focused on the plain database evolution (Curino et al. 2008). Later, the authors extended it to PRISM++, which includes the modification of constraints and update rewriting (Curino et al. 2010) as well as an algorithm to derive the evolution from one given schema version to another as a sequence of SMOs (Curino et al. 2012). To benchmark database evolution languages and tools, researchers also analyzed the evolution histories of open source projects (Curino et al. 2008; Skoulis et al. 2014). Finally, database versioning extends the ideas of database evolution to allow both forward and backward compatibility between the different versions of evolving schemas (Herrmann et al. 2016; Roddick 1995). Another extension of PRISM takes a first step into this direction by answering queries on former schema versions according to the current data (Moon et al. 2009).

CODEL inherits the principle style of PRISM, particularly the coupling of both schema and data evolution in compact SMOs, which facilitates valuable features like e.g. semi-automatic variant co-evolution or database versioning in the first place. However, PRISM is not relationally complete, while CODEL is. This additional characteristic provided by CODEL is highly valuable with respect to the facilitated features, since falling back on common DDL and DML evolution scripts makes them inapplicable. Table 3 summarizes the key differences between traditional SQL based approaches, PRISM, and CODEL. All three are practically complete, hence they can express common evolution scenarios (shown for Wikimedia in Section 3.1). While SQL is relationally complete but separates the schema evolution from the data evolution, PRISM has exactly the opposite

Table 3 Comparison of database evolution approaches

	SQL	PRISM	CODEL
Practical completeness	✓	✓	✓
Relational completeness	✓	✗	✓
Couple data- and schema evolution	✗	✓	✓

characteristics. The major contribution of CoDEL is to combine both characteristics. Even though developers are already familiar with SQL, the additional effort of learning a new language like PRISM or CoDEL pays off in the end, thanks to the facilitated features.

With VaCo, we showcase how to use the power of database evolution languages for semi-automated variant co-evolution. The problem of model co-evolution in general is broadly examined in research, e.g. (Qiu et al. 2013) provides an analysis of the relevance and challenges of co-evolving a database schema and the actual application code. Many researchers aim to automate and simplify this process (Cicchetti et al. 2008; Terwilliger et al. 2010b). However, the evolution of data is typically out of the scope. CoDEL precisely captures the evolution of both schema and data and hence facilitates to semi-automate the variant co-evolution with VaCo. Including the data into the variant co-evolution now greatly simplifies continuously developing multiple components of a software in an agile manner.

6 Conclusion

Agile software development methods embrace the change. While software developers find support in refactoring methods to evolve their software, database developers still have to fiddle with SQL DDL/DML scripts to evolve schema and data of a productive database consistently. Adding evolution support to a DBMS involves the design of a database evolution language (database evolution language). In this paper we considered the relational completeness of database evolution languages for relational databases. Relational completeness is an important property of database evolution languages, since incomplete database evolution languages can force the user back to the manual evolution process based on SQL DDL and DML limiting the utility of the evolution functionality. We presented the relational complete database evolution language CoDEL. We detailed its formal definition and showed its relational completeness. CoDEL is to our best knowledge the first well-defined, relational complete database evolution language. CoDEL can serve as a reference language for productive implementations of database evolution in DBMSes.

The solid formal base of CoDEL is important for research and development beyond database evolution. For instance, we showed how to use CoDEL to semi-automate variant co-evolution with VaCo. As CoDEL SMOs capture the developer's intention both on the schema and data level, VaCo can combine different evolutions. Given a variant that is based on a given core, VaCo proposes a co-evolved variant whenever the core evolves. This greatly supports developers with the expensive and error-prone task of manual co-evolution. Promising future work could extend VaCo with schema constraints that are maintained over variants to give developers guarantees on invariant parts of their variants. Further, CoDEL itself could be extended to cover database constraints—e.g. by mapping the convenient constraint evolution of PRISM++ (Curino et al. 2010) to CoDEL as well. For the near future, we hope CoDEL helps to jump start more implementations of proper database evolution features in DBMSes, so agile development methods finally arrive at the database layer.

Acknowledgments This work is funded by the German Research Foundation (Deutsche Forschungsgemeinschaft; DFG) within the RoSI research training group (GRK 1907).

References

Ambler, S.W. (2006). Whence Data Management?. *Dr. Dobb's Journal*, *390*, 79.

Ambler, S.W., & Sadalage, P.J. (2006). Refactoring Databases: Evolutionary Database Design. Addison-Wesley Signature, isbn 978-0321774514.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., & et al (2001). Manifesto for Agile Software Development.

Ceri, S., Negri, M., & Pelagatti, G. (1982). Horizontal Data Partitioning in Database Design. *SIGMOD Conference*, 128–136.

Cicchetti, A., Ruscio, D.D., Eramo, R., & Pierantonio, A. (2008). Automating Co-evolution in Model-Driven Engineering. *EDOC*, 222–231.

Codd, E.F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, *15*(3), 162–166.

Curino, C.A., Moon, H.J., Deutsch, A., & Zaniolo, C. (2010). Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *VLDB Endowment*, *4*(2), 117–128.

Curino, C.A., Moon, H.J., Deutsch, A., & Zaniolo, C. (2012). Automating the Database Schema Evolution Process. *VLDB Journal*, *22*(1), 73–98.

Curino, C.A., Moon, H.J., & Zaniolo, C. (2008). Graceful Database Schema Evolution: the PRISM Workbench. *VLDB Endowment*, *1*(1), 761–772.

Curino, C.A., Tanca, L., Moon, H.J., & Zaniolo, C. (2008). Schema Evolution in Wikipedia: Toward a Web Information System Benchmark. *ICEIS*, 323–332.

Domínguez, E., Lloret, J., Rubio, Á.L., & Zapata, M.A. (2008). MeDEA: A Database Evolution Architecture with Traceability. *Data & Knowledge Engineering*, *65*(3), 419–441.

Herrmann, K., Reimann, J., Voigt, H., Demuth, B., Fromm, S., Stelzmann, R., & Lehner, W. (2015). Database Evolution for Software Product Lines. *DATA*, 125–133.

Herrmann, K., Voigt, H., Behrend, A., & Lehner, W. (2015). CoDEL – A Relationally Complete Language for Database Evolution. *ADBIS*, 63–76.

Herrmann, K., Voigt, H., Seyschab, T., & Lehner, W. (2016). InVerDa – Co-existing Schema Versions Made Foolproof. *ICDE (Demo)*.

Moon, H.J., Curino, C.A., Ham, M., & Zaniolo, C. (2009). PRIMA – Archiving and Querying Historical Data with Evolving Schemas. *SIGMOD Conference*, 1019–1022.

Papastefanatos, G., Vassiliadis, P., Simitsis, A., Aggistalis, K., Pechlivani, F., & Vassiliou, Y. (2008). Language Extensions for the Automation of Database Schema Evolution. *ICEIS*, 74–81.

Qiu, D., Li, B., & Su, Z. (2013). An empirical analysis of the co-evolution of schema and code in database applications. *ESEC/FSE*, 125.

Rahm, E., & Bernstein, P.A. (2006). An Online Bibliography on Schema Evolution. *SIGMOD Record*, *35*(4), 30–31.

Roddick, J.F. (1992). SQL/SE – A Query Language Extension for Databases Supporting Schema Evolution. *SIGMOD Record*, *21*(3), 10–16.

Roddick, J.F. (1995). A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, *37*(7), 383–393.

Skoulis, I., Vassiliadis, P., & Zarras, A. (2014). Open-Source Databases: Within, Outside, or Beyond Lehman's Laws of Software Evolution *LNCS*, *8484*, 379–393.

Terwilliger, J.F., Bernstein, P.A., & Unnithan, A. (2010). Worry-Free Database Upgrades. *SIGMOD Conference*, 1191.

Terwilliger, J.F., Bernstein, P.A., & Unnithan, A. (2010). Automated co-evolution of conceptual models, physical databases, and mappings. *ER*, 146–159.

Terwilliger, J.F., Cleve, A., & Curino, C.A. (2012). How Clean is Your Sandbox *LNCS*, *7307*(2012), 1–23.

Ullman, J.D. (1988). *Principles of database and knowledge-base systems*: Computer Science Press. ISBN 9780881751888.

Zaniolo, C. (1984). Database Relations with Null Values. *Journal of Computer and System Sciences*, *28*(1), 142–166.

Kai Herrmann is a PhD student at the Dresden Database Systems Group at Technische Universität Dresden as well as an associated PhD student at Aalborg University. After obtaining his diploma degree in Dresden in 2013, he worked for one year in the EuroTRACKex industry project and designed a flexible database layer for a software product line. Inspired by the gained insights, in 2014 he joined the RoSI research training group that promotes dynamic and adaptive software development. His particular research interests are database evolution, database versioning, as well as flexible database design.

Hannes Voigt is a post-doctoral researcher at the Dresden Database Systems Group, Technische Universität Dresden and obtained his PhD from the same university in 2014. He worked on various database topics such as physical design, management of schema-flexible data, and self-adapting indexes. From 2010 to 2011, he worked at SAP Labs, Palo Alto contributing to a predecessor of SAP HANA Graph Project. His current research focuses on database evolution and versioning, declarative graph query languages and efficient graph processing on NUMA in-memory storage systems. He is also member of the LDBC Graph Query Language Standardization Task Force.

Jonas Rausch received his diploma degree in Computer Science from Technische Universität Dresden in 2016. His research focus comprises database evolution as well as database versioning. He is currently working at ORSOFT GmbH, Leipzig where he contributes to innovative and reliable solutions as addition to SAP ERP, SAP SCM and SAP S/4HANA for production planning, advanced planning and scheduling, and supply chain management.

Andreas Behrend is a senior researcher in the Intelligent Database Group at the University of Bonn. He studied Computer Science and Economics in Rostock, Aberdeen, and Bonn. He received a Ph.D. and a habilitation degree, resp., from the University of Bonn in 2004, resp. 2011. He was on leave to the University of Dresden (2014), Marburg (2015), and Halle-Wittenberg (2016) holding an interim full professorship for database systems resp. Big Data Analytics. His research is focused on monitoring applications, in-database analytics, indexing temporal data on NUMA in-memory storage systems, data-driven model generation and visualization as well as deductive and predictive reasoning.

Wolfgang Lehner is head of the Dresden Database Systems Group at Technische Universität Dresden. He obtained his Ph.D. degree (Dr.-Ing.) at the University of Erlangen-Nuremberg. Subsequently, he joined the Business Intelligence (BI) group at the IBM Almaden Research Center in San Jose (CA). After his return to Germany, he again joined the database system group at University of Erlangen-Nuremberg as a senior researcher. From 10/2000 to 2/2002, Wolfgang Lehner was on a temporary assignment at the University of Halle-Wittenberg, holding the professorship for database systems. In 7/2001 he finished his habilitation and was awarded with the venia legendi. Since 10/2002, Prof. Lehner has been conducting research and teaching at TU Dresden and is involved in multiple industrial projects. He was a temporarily visiting scientist at the University of Waterloo (Canada), SAP Palo Alto, GfK Nuremberg, SAP Walldorf, UBS WMBB Zurich, and Microsoft Research in Redmond (WA). Wolfgang Lehner is currently the spokesperson of the Review Board on Computer Science of DFG (German Research Foundation). Since 2014, he is also an appointed member of Academia Europaea (The Academy of Europe).