⟦A⁝B⁝C⟧ Evolution in Database Applications [Vision]

Stefanie Scherzinger*
Technical University of Applied Sciences Regensburg
Germany
stefanie.scherzinger@othr.de

Wolfgang Mauerer*
Technical University of Applied Sciences Regensburg
Siemens AG, Corporate Research, Munich
Germany
wolfgang.mauerer@othr.de

Haridimos Kondylakis
Institute of Computer Science, FORTH
N. Plastira 100
Heraklion, Crete, Greece
kondylak@ics.forth.gr

* Contributed equally

ABSTRACT

Databases are at the core of virtually any software product. Changes to database schemas cannot be made in isolation, as they are intricately coupled with application code.

Such couplings cause *collateral evolution*, which is a recognized, important research problem. In this vision paper, we uncover the aspect of *multivariate collateral evolution* (MCE). This is a wide-spread phenomenon, inevitable in software that supports alternative database backends, since vendor-specific SQL dialects necessitate a simultaneous evolution of both, database schema *and* program code, for all supported *DB variants*. These near-same changes impose substantial manual effort. In this paper, we quantitatively illustrate that MCE is a real-world problem, so far neglected in research, to the best of our knowledge.

We then propose *DeBinelle*, a novel framework and domain-specific language for *multivariate semantic patches* that abstracts DB-variant schema changes and coupled program code into a single, unified representation. DeBinelle offers a novel alternative to manually tackling MCE. It considerably extends established, seminal results in software engineering research for single-domain, single-language semantic patches: DeBinelle caters to several domains (the application logic as well as the database backend), and supports several programming languages, as well as SQL and its many dialects. It effectively eliminates the need to perform vendor-specific changes, replacing them with intuitive multivariate semantic patches. We discuss the benefits of using DeBinelle, based on real-world use cases from reference systems for schema evolution, like MediaWiki. We finally introduce a system architecture, and identify research challenges.

PVLDB Reference Format:
S. Scherzinger, W. Mauerer, and H. Kondylakis. ⟦A⁝B⁝C⟧ Evolution in Database Application. *PVLDB*, 01(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

Proceedings of the VLDB Endowment, Vol. 01, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Decades of research have contributed frameworks for managing database schema evolution, which still remains an open challenge [42]. These frameworks gracefully manage the evolution between schemas, denoted $S \mapsto S$ in Part ① of Figure 1. Early proposals date back to the 80s [38], among the more recent are PRISM [13], InVerDa [21], or CHiSEL [40].

Yet from the perspective of software developers, the schema does not evolve in isolation. Whenever the schema changes, other parts of the code, such as GUIs and application logic, change as well. This coupling causes *collateral evolution*, a known and important challenge [2, 3, 12, 29, 34]. In Part ② of Figure 1, we denote this form of evolution as $(S \times C) \mapsto (S \times C)$, since the schema and program code evolve together.

However, numerous applications support several alternative database backends (or *variants*), which adds complexity: In the reference applications listed in Table 1, proprietary SQL dialects[1] lead to multiple, essentially identical schemas. When these applications are installed, the operations team chooses among one of the supported database variants. Yet during application development, each schema variant is again coupled with code. The joint evolution can be formally described as $\{S \times C\} \mapsto \{S \times C\}$, where sets of schema-code couplings evolve together (illustrated in Part ③). We refer to the challenge of evolving several database schemas, and the dependent code, as *multivariate collateral evolution* (MCE).

The evolution histories of most of the open source applications in Table 1 have been extensively studied as reference systems for schema evolution (however, while few studies consider collateral evolution [34], all studies known to us disregard MCE, for instance [11, 14, 15, 34, 45, 50]). In particular, the de-facto schema evolution benchmark [15] is based on MediaWiki, the software underlying Wikipedia, that currently supports three DB variants. Thus, MCE is a problem affecting an important class of real-world database applications. In this paper, we will use quantitative measurements on real-world systems, to show the extent of this problem.

Existing academic frameworks for schema evolution, and even practitioners' tools (like Flyway), do not address MCE, primarily because they evolve a single schema variant, and therefore only address the most simple case visualized in Figure 1 Part ①. With multiple database variants, they would have to be separately prepared and applied, complicating the evolution process even further.

[1] Wikipedia (link available in PDF) compares SQL dialects.

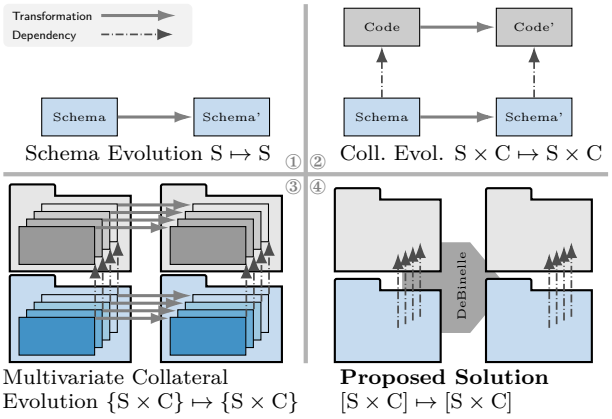


Figure 1: Database application evolution scenarios.

The interdependencies underlying MCE are the focus of this vision paper: While the details of each variant-specific evolution differ, they all represent the *same* conceptual change (*e.g.*, adding a table or changing an attribute type). Our envisioned solution lifts the problem from syntactic details to an abstract change—the set of variant-specific changes if represented by an equivalence class that captures the very core of the change. We formalize this as $[S \times C] \mapsto [S \times C]$, where we evolve the whole class of schema and code couplings, see Part ④ in Figure 1. More specifically, our contributions in this paper are the following:

- We introduce the problem of multivariate collateral evolution, characterized by the need for performing recurring, conceptually identical changes to database schema and program code, for each *database variant*. We argue that this is a real and important problem that has so far been overlooked in research. To the best of our knowledge, this is the first paper dedicated to this problem.
- We present our vision of the DeBinelle framework, introducing *multivariate semantic patches* (MSPs) to support multivariate collateral evolution. DeBinelle is inspired by, and considerably extends, established seminal results in Coccinelle [8].[2] We extend existing work on transformations of a single general purpose language to multiple languages common in database applications.
- We sketch the DeBinelle systems architecture and outline an implementation strategy.
- We provide an extended appendix that shows our vision is feasible, and that provides first details. Previous work relies on temporal logic as the underlying formalism, and for establishing a formal semantics, at the expense of an EXPTIME-complete foundation. We deliberately separate these concerns, and devise an implementation strategy based on robust PTIME mechanisms (embedding directed, acyclic graphs), while we base the formal semantics on Datalog.

Outline. The rest of this paper is structured as follows: In Section 2 we present key insights from a case study on multivariate collateral evolution. Then in Section 3 we sketch our vision of multivariate semantic patches and DeBinelle. Section 4 presents related work. Section 5 concludes this paper, presents success criteria and open challenges. Finally, in Appendix A, we sketch DeBinelle internals and semantics.

[2]In naming our tool, we pay homage to the seminal work on Coccinelle. In playful reference to the syntax of the DeBinelle language that we propose in this paper, we refer to our approach of handling MCE as $[\![\mathrm{A} :|: \mathrm{B} :|: \mathrm{C}]\!]$ evolution.

Project	Domain	DB Variants Supported	Used in
BiblioteQ	Library	PgSQL, SQLite	[50]
Joomla!	CMS	Azure SQL, MySQL, PgSQL	[34]
MediaWiki	Wiki	MySQL, PgSQL, SQLite	[14, 15, 34]
Oscar	Hospital	MySQL, Oracle, PgSQL	[11]
PowerDNS	Name Srv.	MySQL, Oracle, PgSQL, SQLite	
RoundCube	Webmail	MSSQL, MySQL, Oracle, PgSQL, SQLite	[34]
Zabbix	IT Monitor	DB2, MySQL, Oracle, PgSQL	[14, 45]

Table 1: High-profile MCE applications and their supported DB-variants (PgSQL = PostgreSQL).

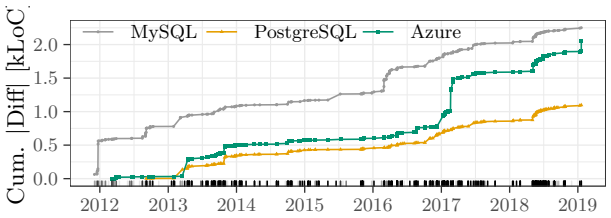


Figure 2: Cumulative number of LoCs changed in incremental SQL diff files, resolved by time.

2. KEY INSIGHTS

We next introduce multivariate collateral evolution based on a real-world use case. We make a point that existing solutions do not sufficiently address this problem.

2.1 Multivariate Collateral Evolution: An Unsolved Real-World Problem

Collateral evolution in database applications describes a known challenge, and has been studied in the past [12, 29, 34]. However, real-world development faces even more complex challenges. We therefore dedicate this section to presenting our insights on *multivariate* collateral evolution.

Real-World Examples. There is a practical family of database applications that support alternative DB variants. In Table 1, we list open source applications, where most are well-known from literature, either from empirical studies on schema evolution or used for benchmarking schema evolution frameworks (always focusing on $S \mapsto S$ evolution). We state the project name, its purpose, and list the DB variants supported.[3] In the source code repositories of all these applications, there are multiple, DB-variant schema declarations, that is, individual data definition language (DDL) files[4].

In the following, we focus on the open source application Joomla!, a well-known content management system. Currently, Joomla! maintains three DB-variants of `joomla.sql`, the file declaring the database schema.

[3]This is the state as of Jan-2020, since different systems may have been supported in the past. For instance, MediaWiki used to support Oracle and MSSQL.

[4]Note that there are also dedicated frameworks abstracting from DB-variant schemas (*e.g.*, XMLDB editor). Nevertheless, by our experience, the source code in such applications still contains conditionals distinguishing DB variants.

Table 2: Normalised similarities between cumulative change size time series for the main schema files (see Figure 3 for the underlying time series). Point-wise identical time series reach 100% similarity.

	2013	2014	2015	2016	2017	2018
MySQL/PgSQL	89%	62%	59%	89%	77%	88%
MySQL/Azure	92%	55%	62%	89%	85%	89%
PgSQL/Azure	84%	68%	58%	81%	78%	85%

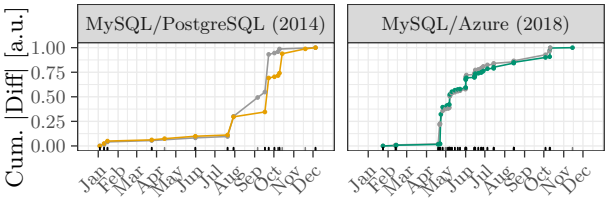


Figure 3: Exemplary excerpts of time series (normalised to identical codomains) for the cumulative size of changes to variants of the main schema file.

Maintenance Effort over Time. To illustrate the maintenance effort for evolving schema files, we compute the size of each *incremental schema update*, measured in lines of code (LoC), and accumulate the values resolved by database variant and over time in Figure 2. Starting in 2012, only MySQL and Azure SQL are supported, followed by the addition of PostgreSQL. Thus, three near-identical versions of the schema declaration evolve over time. The effort adds up to 2k lines of code edited for schema changes on MySQL alone, a substantial effort. It is apparent from the graph that one conceptual change replicates to all DB variants.

To quantify the coupling, we pair-wise compare the growth time series in Figure 2 for all DB variants using the *normalised compression distance* [9]. The measure, computed in yearly granularity, enjoys a sound theoretical foundation based on Kolmogorov complexity, but can be readily computed numerically [31]. Table 2 shows the agreement between the DB-variant schemas (100% implies point-wise identical growth curves). Usually, evolution proceeds with high similarity. This once more underlines duplicated efforts that steadily grow with the number of DB variants.

Figure 3 provides intuition on the similarity measure by connecting some of the percentage values in Table 2 with the underlying main schema evolution graphs: The agreement among MySQL and PostgreSQL in 2014 is only 62%, as can be seen by the slightly differing time series, whereas MySQL and Azure SQL exhibit a very synchronous pattern in 2018.

Although we restrict this discussion to Joomla!, the observations generalize to other applications listed in Table 1.

Analyzing a Single Schema Change. We next point to one specific, real-world example that highlights a typical pain point. Figure 4 shows changes from July 10th, 2019 (commit hash 6b92cc3). The type of attribute `home` from a specific table is changed from numeric to string, after developers realized that the domain includes strings.

The figure shows *syntactic* modifications that change the type of an attribute, for all of the three DB variants. While the changes are very similar, both the old and the new attributes types are DB-variant. This change is accompanied by a discussion thread that involves nine developers, and is carried on over the course of 50 days, until the changes are finally committed to the code base. In particular, developers debate the difficulty of testing the changes' correctness, as well as implications on other parts of the code (such as a custom-written, programmatic schema checker tool used to verify that schema updates have been correctly applied to the production database). Thus, type changes can be far from trivial. As we discuss next, they seem to be particularly common in MCE scenarios, yet are not supported by academic schema evolution frameworks.

2.2 Limitations of Existing Tools

In the following, we describe how syntax-based tools could be used for basic matching tasks. We show that this can be surprisingly involved, to the point where manual changes may be easier to apply. Then, we explain the shortcomings of existing schema evolution frameworks when it comes to multivariate collateral evolution.

Limitations of Syntax-based Tools. The three changes required in Figure 4 can be generalised to a *single* syntactic transformation by several means. A straightforward pragmatic approach would be to update the schema declarations using textual transformations (eg, `sed` or other popular tools). Yet to just *match* the desired statements for MySQL, a complex regular expression of some 500 characters (not shown) is required. Such low-level scripts are difficult to read (often for anyone else but, sometimes even including their author), maintain, and get right. Syntactic disadvantages of regular expressions include issues of ordering (`NOT NULL DEFAULT 0` is equivalent with `DEFAULT 0 NOT NULL`) and scoping (for instance, restricting changes to a specific table), and technical nuisances (handling multi-line expressions, different capabilities of different implementations, issues with mixing upper and lower case spelling, ...) Such seemingly simple issues create practical inconvenience. Unambiguously specifying the allowed input for a DB variant, and mapping the detected strings to proper DB-variant output, also integrates badly with regular expressions.

In programming, it is generally acknowledged that refactoring transformations—down to trivialities like renaming variables [46]— that look deceptively simple, become unwieldy, if expressed in general transformation languages [39]. Combining programming languages with the intricacies of SQL dialects does not necessarily alleviate the situation.

Shortcomings of Schema Evolution Frameworks. All schema evolution frameworks known to us focus on evolving a single schema variant only. That is, they are not designed to handle multivariate collateral evolution. Moreover, academic frameworks typically support complex *schema modification operations (SMOs)* on the logical schema, such as merging or splitting relations. Yet empirical studies show that basic changes are actually the most frequent for applications considered in Table 1. In the prominent open source projects Joomla!, MediaWiki, and Roundcube, changing a type is the most frequent schema change, outweighing even table or column creation [34] . Similarly, in [50] it is shown that in BiblioteQ, type changes constitute more than half of all schema changes, more than twice as table or attribute creation combined. As mentioned in Section. 2.1, it seems that attribute type changes may have been traditionally underestimated with respect to their impact on the application code, which may be why they are generally not addressed by academic frameworks.

```
@@ -1801,7 +1801,7 @@                    @@ -2575,7 +2575,7 @@                    @@ -1831,7 +1831,7 @@
  "id" serial NOT NULL,                          "id" bigint IDENTITY(9,1) NOT NULL,          `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  "template" varchar(50) DEFAULT '' NOT NULL,    "template" nvarchar(50) NOT NULL DEFAULT '', `template` varchar(50) NOT NULL DEFAULT '',
  "client_id" smallint DEFAULT 0 NOT NULL,       "client_id" tinyint NOT NULL DEFAULT 0,      `client_id` tinyint(1) unsigned NOT NULL DEFAULT 0,
- "home" smallint DEFAULT 0 NOT NULL,          - "home" tinyint NOT NULL DEFAULT 0,         - `home` tinyint(1) unsigned NOT NULL DEFAULT 0,
+ "home" varchar(7) DEFAULT '0' NOT NULL,      + "home" nvarchar(7) NOT NULL DEFAULT '0',   + `home` char(7) NOT NULL DEFAULT '0',
  "title" varchar(255) DEFAULT '' NOT NULL,      "title" nvarchar(255) NOT NULL DEFAULT '',   `title` varchar(255) NOT NULL DEFAULT '',
```

Figure 4: DB-variant DDL code in Joomla! (commit 6b92cc3), changing the type of attribute `home` in a `create table` statement by a traditional syntactic patch for PostgreSQL (left), Azure SQL (inner), and MySQL (right) in file `installation/sql/⟨variant⟩/joomla.sql`. Lines starting with a minus symbol are removed, lines with a plus symbol are added. *Hunk headers* "`@@` $-l_1, \Delta_1$ $+l_2, \Delta_2$ `@@`" identify a line l_1 within the source file where code (Δ_1 lines long) is removed, and line l_2 where code of length Δ_2 is inserted.

3. ENVISIONED SOLUTION

We next motivate semantic multivariate patches and sketch the elegant (and powerful) solutions we envision for DeBinelle. We outline the DeBinelle systems architecture, and provide a first impact assessment based on real world use cases.

3.1 Multivariate Semantic Patches

Syntactic transformation might convey the impression of primitive replacements, but they are common artefacts of global and incremental software development. Consequently, software developers are used to working with syntactic patches (large, influential development projects like the Linux kernel discuss, refine and integrate more than 5,000 patches per month [35]). We follow Coccinelle [8] for the *style* of the domain-specific language underlying DeBinelle, but considerably extend its expressiveness, semantics and functionalities to multi-DB, multi-language scenarios.

DeBinelle understands the semantics of SQL (and the many specific dialects) and programming language code. This allows us to condense related syntactic changes into one *single* representation—a semantic patch—from which multiple transformations derive. DeBinelle can even handle SQL statements assembled as raw strings within the application code, as often found in legacy database applications. This might be seen as a superficial technical detail or as a discouraged programming technique that is inconsequential to the database community. It is rarely addressed in the literature, or by any schema evolution framework known to us. Nonetheless, it occurs in many real applications, including very high-profile ones such as Joomla! and MediaWiki.

Abstracting from syntactic patches to describe schema and code evolution is possible in various ways. Assuming the viewpoint of category theory [18], a database can be seen as a functor that maps a schema category to a set representing data. Schema transformations are then represented as natural transformations, and the introduction of multiple variants adds another categorical layer. While such a point of view (that we will adapt in the eventual realisation of DeBinelle) is useful in many ways, it *over*-generalises for practical applications as much as purely syntactic transformations suffer from a *lack* of generalisation. We design DeBinelle to inhabit the middle grounds between these extremes: Practical utility should be combined with a solid theoretical underpinning, and just the right amount of abstraction.

3.2 Examples

We begin with small examples that familiarize readers with DeBinelle notation(the full grammar for the DeBinelle language is provided in Appendix A.1. We then proceed with a more complex example that shows the capabilities of multivariate semantic patches.

3.2.1 First Examples

Let us consider the changes to table declarations shown as syntactic transformations in Figure 4: The attribute `home` is renamed to `user_home` for each DB variant. Developers are familiar with such syntactic patches, but while they are useful for day-to-day development, they do not allow for much generalization.[5] The same change, on the other hand, is uniformly expressed for all variants by the single DeBinelle patch in Figure 5.

The syntax resembles syntactic patches. The rule header (starting line 1) declares the patch name, surrounded by the markers `@...@`. The header declares *meta-variables*, where `T` matches some SQL type and `C` matches some constant. Starting with line 5, we specify the transformations. The minus symbol in front of a line states that matching lines will be removed. Lines leading with a plus are added. In this case, the name is changed, but attribute type and default value remain the same.

```
1 @ rename_home @
2 sqltype T; constant C;
3 @@
4
5 - home T NOT NULL DEFAULT C,     /* Find+remove this pattern */
6 + user_home T NOT NULL DEFAULT C, /* ... replace by this.     */
```

Figure 5: DeBinelle patch renaming attribute `home`. The arrows denote the binding of meta-variables.

Note that in Figure 4, the left variant first mentions the default value, and then declares that attributes are not nullable. For the other DB variants, this is declared the other way round. This is relevant for a syntactic transformation, but doesn't need to concern the author of a semantic patch, since DeBinelle has an understanding of SQL syntax and semantics, and abstracts from these low-level details.

We apply the patch to the target code and check whether all three instances of the schema declaration file `joomla.sql` are updated correctly. Actually, in our example, we would realize that table declarations that are part of PHP unit tests have also been updated (a task that we might have even overlooked). In fact, detecting similar patterns throughout

[5] Revision control systems, regardless of the system-specific representation, view changes to code and schema as operations on text (syntactic patches). It is considered best practice to version control all database artefacts [3], including textual schema declarations. Should schemas be maintained as binary artefacts inside some modelling tool, a textual representation can always be obtained.

the code base is the strength of DeBinelle. However, if we should find that the patch is too unspecific, we can restrict its scope, as illustrated next.

Figure 6 shows a DeBinelle patch carrying out the changes from Figure 4, where the type of attribute `home` is changed to a DB-variant seven character string.

```
1  @ map_string7 @
2  sqltype oldtype;
3  @@
4
5  CREATE TABLE #__template_styles (
6     □
7  -  home oldtype NOT NULL DEFAULT 0,
8  +  home @n_string(7) NOT NULL DEFAULT '0',
9     □
10 );
```

Figure 6: DeBinelle patch for Figure 4.

In the rule header, we declare that meta-variable `oldtype` matches an arbitrary SQL type. Starting with line 5 in the patch, we describe the table to match, so all changes are restricted to this scope. Ellipses □ match uninteresting passages. With line 7, DeBinelle matches the original declaration of attribute `home`. DeBinelle will replace this according to line 8, and change the default value to `'0'`. The new type depends on the SQL dialect from the input target code. We assume that the a-priori knowledge, for which DB variant to produce the output, is available (e.g., from the file path, as specified in a configuration file). In a later example, will extract this information from the target code.

The DB variant is an implicit input parameter for the output isomorphism *`n_string(7)`*, which produces the correct DB-variant type (the preceding "`@`" serves as an escape symbol.) We next explain this mechanism, which—at least syntactically—resembles a function call.

3.2.2 Variants and Isomorphisms

We distinguish between two (deliberately separate) classes of *input* and *output* isomorphisms. The former are a concept introduced in existing work on Coccinelle, the latter are a novel contribution of DeBinelle.

Input Isomorphisms. Input isomorphisms collect a set of keywords that can be matched against a given piece of code—their syntax has already been introduced in the above patch `map_string7`. Such isomorphisms are used in automatic program transformation tools [33] to identify functionally equivalent programming language constructs, for instance `x==NULL`, `!x`, and `NULL==x` in C. The problem in our case extends from *intra*-language equivalences to syntactically different, yet semantically identical statements *across* SQL dialects. A comprehensive collection of input isomorphisms will be provided with DeBinelle as a standard library.

Beyond the above examples, input isomorphisms can be restricted to apply to certain contexts: For instance, the following transformation is only applied on SQL function calls in the target code, as denoted by <`SQLFunction`>:

```
input-iso <SQLFunction> @ stringlen @ expression E; @@
  LENGTH(E) <=> CHAR_LENGTH(E)
```

The input isomorphism *`stringlen`* states that when applying a patch, `LENGTH(E)` and `CHAR_LENGTH(E)`, where meta-variable `E` matches an expression in a SQL query, are considered equivalent. Since input isomorphisms are named, we can disable them individually, should this be required.

The following input isomorphism treats table declarations with the restriction `IF NOT EXISTS` the same as if this restriction were not present.

```
input-iso @ ignore-ifne @ identifier T; expression E; @@
  CREATE TABLE T IF NOT EXISTS (E); <=> CREATE TABLE T (E);
```

Output Isomorphisms. Output isomorphisms allow us to map (possibly) DB-variant input to DB-variant output; this creates multiple syntactic transformations, respectively changes different portions of a target language file induced by the *same* multivariate semantic patch. DeBinelle infers the DB variant from context, and provides it as parameter to every output isomorphism.

```
output-iso @ n_string @ integer N; @
variant mysql      => char(N)
      | sqlazure   => nvarchar(N)
      | sqlite     => text
      | _          => varchar(N) // For all other DB variants.

output-iso @ date_type @ @@
variant sqlite     => text       // SQLite has no date type.
      | _          => date

output-iso @ read_date @ expression E; @@
variant sqlite     => date(E)
      | _          => E // This assumes E has SQL type date.

output-iso @ today @ @@
variant sqlite     => date('now', 'localtime')
      | mssql      => cast(getdate() as date)
      | _          => current_date
```

Output isomorphism *`n_string`* can be used to replace the aptly matched input by an SQL data type that stores a string of n characters. It is used in the patch `map_string7`.

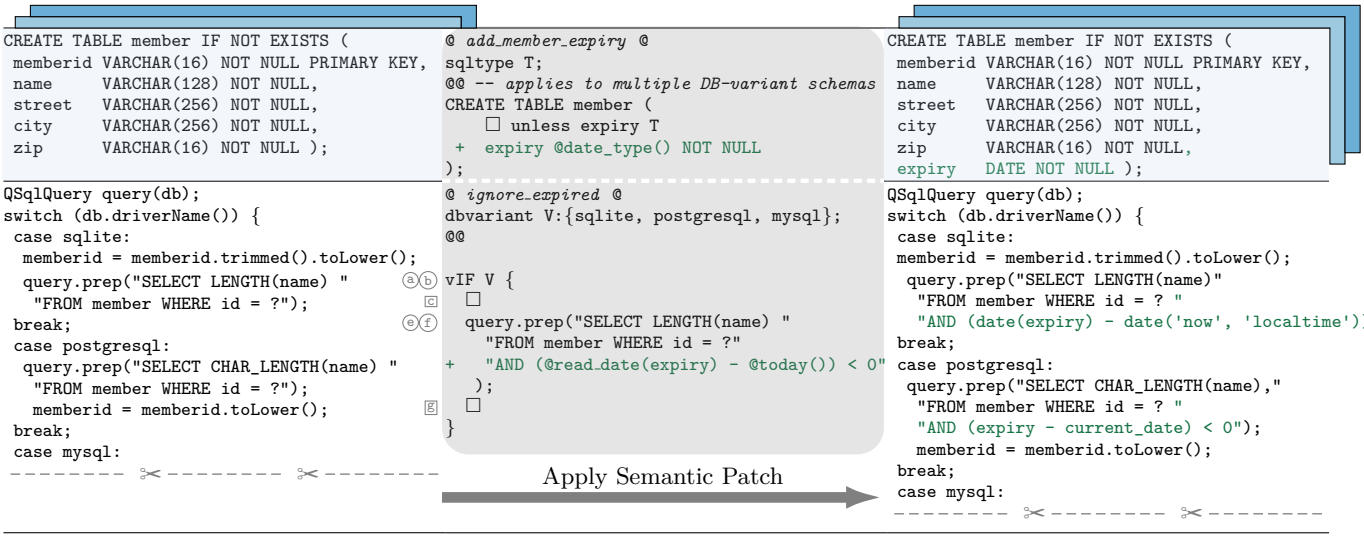
Output isomorphism *`date_type`* is a placeholder for a date type. In SQLite, there is no native date type, so for this DB variant, date values are stored as `text`. We will shortly see an example using this isomorphism.

Beyond the implicit variant argument that is always present, an output isomorphism can have *explicit* parameters: Output isomorphism *`read_date`* assumes `E` is a date-typed expression (*e.g.*, a date attribute value read from a table), unless the DB variant is SQLite, in which case the value needs to be cast into a formatted date first. Output isomorphism *`today`* returns the current date. Albeit both functionalities are achieved with different means in SQLite, the patch author need not bother with these specifics.

When DB variants require specific individual treatment (which, usually, happens in the database access layer code), implementations execute DB-variant target language code. Details of the available means depend on the target language, but we distinguish between three generic classes of mechanisms: a) The usual control-flow statements (`if...then`, `switch...case`, ...)—each branch is devoted to one particular variant; b) inheritance and polymorphism in object-oriented languages—each variant is handled by a derived class, and c) file-level variants—one or more files are dedicated exclusively for each DB variant. We discuss the technical details of matching DB variants in the Appendix A.3.

3.2.3 A Complete Example

In Figure 7, we show a more involved example, from a hypothetical, BiblioteQ-like library management software. The C++ code (bottom left) is not an example of good coding



(a) Input Schema/Target Code (b) DeBinelle Semantic Patch (c) Output Schema/Target Code

Figure 7: An MCE scenario, based on real-world patterns observed in large projects. Attribute `expiry` is added to relation `member`. Below: Queries are adapted. In both cases, one of *many* possible matching code snippets is shown on the left. The multivariate semantic patch comprising two parts (middle) produces the output on the right for the given target code (left). The scissors mark code not shown. (Circles and squares in light colour refer to nodes in Figure 11, see Appendix).

practices (*e.g.*, there is a switch-statement without default clause, and no error handling at all). However, this example unifies several patterns that we have repeatedly observed in real-world projects (including the projects in Table 1, but also in industrial closed-source projects). We distinguish three DB variants, to keep the example manageable.

The upper half of Figure 7 shows the addition of a new attribute `expiry` to the relation managing member data. To the left, we see an excerpt from *one* DDL file, out of many variants. Top center, we see the patch appending the new attribute to the end of the table declaration. The ellipsis operator □ combined with `unless` ensures the patch only matches table declarations where this attribute does *not* (yet) exist (`T` denotes some SQL type). This prevents a redundant introduction, should the attribute already exist.

Note that the patch can be applied to the input, even though lines 4 in the patch, and line 1 in the table declaration, differ in the restriction `IF NOT EXISTS`. This is the effect of the input isomorphism *`ignore-ifne`* introduced earlier.

The closing parenthesis on line 7 is the anchor for inserting the new attribute. DeBinelle is SQL-aware and automatically inserts a new comma, separating `expiry` from the preceding attribute. SQLite does not implement a native date type, yet the other DB variants do. We have the output isomorphism *`date_type`* (introduced before) insert the correct type, based on the DB-variant known from the context (here, the file path). To the right, we show the modified target code.

This schema change is coupled with further code changes, and we focus on one specific task: In the original target code shown to the bottom left, a SQL query retrieves the length of library member names. This query is assembled from raw strings. In line 2, the control flow diverges: Switch labels distinguish between DB variants.

Bottom center, we show the DeBinelle patch. Let us assume that we need to adapt all such SQL queries accessing the `member` relation, so that only active memberships are considered. We therefore change the where-condition accordingly. (Again, since DeBinelle is aware of SQL syntax and semantics, it can automatically insert a single space at the end of the old where condition, so that applying the patch produces a correct SQL query string.)

Line 2 of the semantic patch introduces variant anchors, for MySQL, SQLite and PostgreSQL. Then, the DeBinelle conditional `vIF` matches conditionals in the target programming language that introduce variant scope. For instance, in the input target code, the C++ switch statement introduces variation points for SQLite (lines 3–7), PostgreSQL (lines 8–12), and also MySQL (where the scissors mark code not shown). Thus, `vIF` abstracts from imperative conditional statements (and also matches this `switch...case` statement).

Note that the input isomorphism *`stringlen`*, introduced previously, allows for matching `LENGTH(name)` from the patch against both lines 5 *and* 9 in the input target code, even though they differ syntactically. Applying this input isomorphism to the patch introduces a disjunction that allows for matching any of the possible alternatives. This changes line 7 in the patch as follows (illustrated in the expansion of input iso in Figure 8):[6]

```
[[    query.prep("SELECT LENGTH(name)"
   :|: query.prep("SELECT CHAR_LENGTH(name)" ]]
```

SQLite does not support a native date type, but provides built-in functions for casting strings to dates, and for date arithmetic. The output isomorphisms *`read_date`* and *`today`* from before handle this. Line 9 thus computes the membership expiration date.

[6] Again, we assume DeBinelle to be aware of SQL syntax and semantics, so that it can robustly insert single spaces at the end of a query string, to prevent patches from breaking the syntax of SQL queries.

Variant anchors, a special class of input isomorphisms, identify blocks in code that need to be tailored, depending on the DB variant. The statement `variant V:{sqlite, postgresql, mysql}` in line 2 of the patch associates a *variant anchor set* —keywords that appear in the target language source code—with a meta-variable `V`. When matched in the target code, DeBinelle can infer the DB variant for a block of code, and use this information when expanding output isomorphisms. Multiple variant anchor sets are required when a project uses different labels for such anchors (*e.g.*, constants for `switch` statements, and substrings in class names).[7]

To the bottom right, we show the output target code.

3.3 The DeBinelle Systems Architecture

We have designed a systems architecture, as shown in Figure 8. Structurally, DeBinelle is a variant-aware, multi-language source-to-source transpiler: Given one or more multivariate semantic patches and a set of input files in the target languages, the result is a set of syntactic transformations for each input file. Following usual patterns in compiler construction [20], these transformations are executed on intermediate program representations based on control flow graphs (CFGs). While the domain specific language (DSL) syntax of a multivariate semantic patch is parsed by custom means, we will resort to standard software components (*e.g.*, gcc, clang, Apache Derby) from practical software engineering to prepare full control flow graphs, which DeBinelle then simplifies. Input isomorphisms are expanded on the CFG of the multivariate semantic patch. Then, the patch CFG is embedded into the graph target code CFG. The matching detects *variation points* that require different treatment depending on the DB variant. The final phase performs any replacements (additions and/or deletions)[8] mandated by the multivariate semantic patch and applies output isomorphisms to generate different syntactic changes for each relevant variant. We discuss further details in Appendix A.2.

3.4 A First Impact Assessment

In Table 3, we point out further use cases, identified by manual inspection of the commit histories of selected open source projects. The list is by no means exhaustive, but gives a first sense of scale: We briefly describe each task, and compare the number of lines changed manually (ignoring parallel tasks submitted along with it) with the lines a DeBinelle patch body would require.

For all use cases, less than ten lines of patch code save a substantial amount of repetitive, manual labor. For use case 6, this is not immediately obvious: While there are only 20 lines to change manually, finding these lines is the challenge: A developer has to visually inspect 60 CREATE TABLE statements *per DB variant* (amounting to near 240 table declarations at that time in MediaWiki development) to detect where a unique index can be removed by declaring a primary key instead. Thus, the benefit here is the time a developer spends searching for where to change code, rather than writing 10 lines of patch code.

[7]This capability would allow us to also address more general variability aspects that go beyond databases, although we do no explore this aspect in this paper.

[8]It is possible to specify MSPs that lead to conflicting actions at this stage—for instance, inconsistent additions to one given line—, which can be automatically detected. DeBinelle can then prompt for rephrasing, to avoid ambiguities.

Table 3: Real-world instances of evolution in MCE applications, LoC approx. changed manually (aLMC) vs. LoC in DeBinelle patch body (LPB).

Project/Commit: Description	aLMC	LPB
① BiblioteQ/`2034f41`: Add attribute in SELECT clauses of SQL queries, managed as C++ strings.	120	< 5
② BiblioteQ/`bebc62f`: In deprecating support for MySQL, replace the pattern "`if (mysql) query1 else query2`" in C++ code by else-clause, 25 times.	650	< 5
③ BiblioteQ/`0826815`: Update C++ query strings: If flag `casesensitive` is set, change WHERE predicates with LIKE, casting operands to lowercase.	70	< 5
④ MediaWiki/`2ce92e9`: Remove table `tag_summary` (CREATE/ALTER/DROP), and any indexes on this table (CREATE/ALTER).	70	1
⑤ MediaWiki/`c2bd4b1`: Add new DB-variant table.	50	< 10
⑥ MediaWiki/`267d99f`: Convert the combination of CREATE TABLE and UNIQUE INDEX to a primary key. Affects only 3 files, but requires inspecting over 240 CREATE TABLE statements.	20	< 10
⑦ Joomla!/`2a3fa5f`: Fix integer-typed attributes in CREATE TABLEs that have string-typed defaults.	30	< 5

4. RELATED WORK

As we introduce a new problem in this paper, we report on closely related fields of work.

Empirical Studies on Schema Evolution. MediaWiki and Joomla! are well-studied open source projects (see also the last column in Table 1). Empirical studies on these projects tracked the evolution of the database schema, via the number of tables and columns growing over time, or the nature of schema changes (adding vs. dropping, renaming, etc.). The prominent PRISM schema evolution benchmark [13, 15] is actually based on the real-world commit history of MediaWiki. Notably, all studies known to us focus on a single schema variant changing over time.

There is also dedicated literature for practitioners, discussing schema evolution best practices [2, 3], emphasizing the difficulty of this task due to the coupling of the database schema with application code, but also with the database itself, such as the coupling of tables with stored procedures [2]. Interestingly, the aspect of several DB-variant schemas evolving is not addressed there either.

In addition, there are efforts to capture the impact of a schema change on the remaining application code [34], and to thereby quantify the costs. As a prerequisite, couplings between the database schema and the application source code should be detected [12, 29, 34]. However, we are not aware of any studies on multivariate collateral evolution.

Database Evolution Tool Support. A summary of the various tools, supporting database evolution, is shown in Table 4 for both the practitioners and the academic community. For practitioners, Flyway, Liquibase, Rails Migrations, and DBmaestro Teamwork enable regulated migration between schema versions, supporting all changes in data definition and modification language (DDL & DML) files (with the exception of Rails Migrations, not supporting stored procedures), as shown in Table 4. Most of them are able to write small programs to perform the evolution at the schema level (refer to "Schema patches" in Table 4). Unfortunately, none of these tools support collateral evolution or the even more challenging MCE.

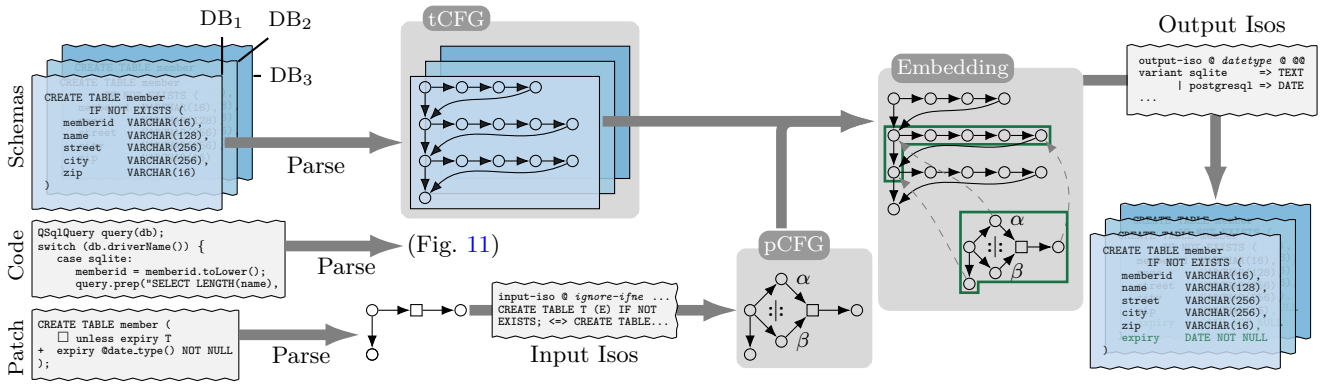


Figure 8: DeBinelle systems architecture and data flow. Interplay between the various standard and custom components, structural representation, and transformations discussed in the Appendix, Section A.2. (The transformation process for C++ target code is separately illustrated, see in Figure 11 in the Appendix.)

Table 4: Comparison of schema evolution tools. We mark features addressed with ✓, and ✗ otherwise.

	SQL	Flyway	Liquibase	Rails Migr.	DB Maestro	Model Mngmt	PRISM	PRIMA	ScaDaVer	Sym. Lense	InVerDa	VESEL	MIGRATOR	CHiSEL	DeBinelle
SMOs	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✗	✓	✓
Integrity constraints	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓
Default values	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓
Types	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Stored procedures	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Schema patches	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
Auto data migration	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✗
Bidir. data migration	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Auto query rewriting	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗
Collat. evolution	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
MCE	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

From the research community, there have been various contributions for single database schema evolution. For example, model management [7] handles multiple schema versions by allowing to match, diff, and merge existing schemas to derive mappings between these schemas. The derived mappings can be used to rewrite old queries or to migrate data. PRIMA [32] introduces simple and intuitive schema modification operations (SMOs) for describing schema evolution, enabling users to issue queries over one of the versions, which are then rewritten to the original schema version to be answered. Capitalizing and extending those SMOs, PRISM and PRISM++ [13] enable schema and data migration, as well as automated query rewriting between the versions. ScaDaVer [47] proposes a schema and data versioning system, based on simple SMOs, exploring alternatives for handling queries against the various branches. Symmetric Lenses [23] facilitates read and write access along a bidirectional mapping, while auxiliary tables persist the complements to not lose any data. InVerDA [21, 22] enables co-existing schema versions in a single database, coupling the evolution of both the schema and the data in intuitive and compact SMOs. Based on those SMOs it is able to propagate data both forward and backward between all schema versions (refer to "Bidirectional data migration" line of Table 4). CHiSEL [40] on the other hand, proposes a wider set of high-level SMOs to simplify complex schema evolution tasks for scientific applications. VESSEL focuses only on explaining schema evolution [4], reusing SMOs from InVerDA, whereas MIGRATOR [48] automatically rewrites update SQL statements upon the schema changes. However, despite the numerous approaches,to the best of our knowledge, there is no academic schema evolution framework handling type changes (refer to Table 4), and most do not handle default value changes either. Yet as discussed in Sec. 2.2, type changes have been shown to be frequent, for at least 5 out of the 7 projects listed in Table 1.

In a way, DeBinelle can be seen as complementary to these tools, as it currently is not designed to support automated migration of existing data, or automatically rewrite SQL queries between schema versions—although it is able to assist in changing the variant-specific, textual representations of SQL queries, as shown in Figure 7. On the other hand, as shown, none of these tools support collateral evolution. With DeBinelle, we can rewrite the multivariate specifications of integrity constraints, default values, or types and stored procedures. Most importantly, DeBinelle is the first tool, to the best of our knowledge, that even targets MCE.

Software Evolution Tool Support. Eliminating manual labour and recurring, repetitive tasks by automation is an intensively studied question in contemporary software engineering research. Software changes are actual *transformations* of programs. They are required in many engineering scenarios — APIs modifications, back-porting of fixes, changes in data structures. Semantic patches have already been employed to declare change *mechanisms*, instead of directly executing the change. For example, the Coccinelle [33] tool lifts patches from a purely syntactic transformation to a (single) semantic meta-description from which (multiple) syntactic patches can be generated for C, and for limited subsets of C++ and Java [8, 27], but lacks support for generating variant output. The matching process is based on an extension of computational tree logic [10, 24], which is a powerful basis for formal proofs, but turns adding support for new languages into a highly non-trivial venture, especially

when the languages are not rooted in the C family. Coccinelle's semantic patch language is *more* powerful than our proposal. In designing the syntax of DeBinelle patches (see the EBNF grammar in Appendix A.1), we have deliberately traded features for substantially reduced implementation complexity.

There is a strong connection between temporal logic and Datalog (*e.g.*, see [1]). In choosing a formal basis for DeBinelle, we found that Datalog is a more natural fit for the database research community, and also, that the translation from patches to Datalog queries can be expressed succinctly.

Although numerous research works focus on refactoring and re-engineering programs (for a review, see [6]) and try to address the issue of optimal evolution over time, consideration for databases is merely non-existent in this community. Databases are usually only mentioned as (yet) another component that is addressed via an abstraction layer, disregarding the problem as we have presented it here. The many issues seen in real-world code that we have discussed in this paper are also often overlooked in these domains.

5. DISCUSSION AND CONCLUSION

In this paper, we focus on the problem of collateral evolution caused by the coupling of schema changes with application code and we introduce, for the first time, the problem of multivariate collateral evolution. We show that this is an important problem, currently requiring a vast amount of time from developers in order to maintain and evolve DB-variant schemas and code. We tried to quantify this effort, showing that there is currently no tool support for such complex, yet common scenarios. Moving one step further, we presented our vision of DeBinelle, an academic tool for applying multivariate semantic patches, explaining how DeBinelle patches will work, and demonstrating several interesting scenarios and use cases. Finally, we have presented our ideas for the architecture of DeBinelle. In the Appendix, we describe potential formalisms behind it.

The entry level to using DeBinelle in an existing project is low: Unlike other tools for schema evolution management (*e.g.*, Flyway) that must be used from the get-go, DeBinelle is not tied to any particular development environment or process, and can be easily adopted by developers already working on an established project. DeBinelle does not require a coordinated, project-wide policy decision, and does not cause technological lock-in. We believe this will motivate developers to engage with DeBinelle. Moreover, DeBinelle can be gradually adopted. Once developers have had one or two successful experiences using DeBinelle, we assume they will grow steadily accustomed to writing code that lends itself nicely as a target for semantic patches. We hypothesise, based on Ref. [28] and our own experience as professional software engineers, that DeBinelle-friendly code will lead to more uniformly structured source code, improved readability, and thus better overall code quality and maintainability.

Success Criteria. The success of DeBinelle can be measured by several criteria that address to different time frames.

A crucial criterion is *usability*. The interface must fit the developers' conceptual model, whereas important, real-world challenges should be elegantly addressed. Developers also are accustomed to thinking about change in terms of increments (fuelled by agile practises) and commits to a version control system (fueled by omnipresent tools like `git`), rather than high-level schema modification operations, as a database administrator would. Developers are accustomed to IDEs, debuggers, provenance visualisation, etc., and DeBinelle should be easy to integrate with said tools, at any stage of the database application evolution. DeBinelle can also be used selectively: Only on DDL or DML statements, or application code—or all of them. Usability can be easily quantified using standard usability questionnaires and the feedback collected will guide further development.

Besides matching the developers' habitus, it should also *integrate with the existing state-of-the-art* in schema evolution research, like high-level SMOs and automated query rewriting, promoting further research in the area. As already identified in the related work section, DeBinelle can be seen as complementing those academic tools. Beyond the initial design presented in this paper, we need to ensure that DeBinelle has a *well-founded syntax and semantics* that is powerful enough to cover a wide range of common use cases—this can be quantitatively measured by describing standard elements of schema evolution benchmarks in the DeBinelle DSL (continuing the work from Tab. 3). Here, the established MediaWiki benchmark [15] is a good fit, since MediaWiki is affected by multivariate collateral evolution. Thus, we can make sure that historic changes in real-world projects can be well expressed. The language design should continue to follow the golden rule of software ergonomics (the principle of least surprise [49]), which can be ascertained by structured developer interviews and surveys.

In the long run, DeBinelle's success rests on *user adoption*. This requires building a community [5, 25, 30] from early on, and aims at establishing a close interaction between users, academics, and developers. We will develop and publish DeBinelle under an open source license and contribution structure (no copyright assignment) that provides appropriate incentives (most importantly, public credit and research opportunities) to academics for extending the system and contributing improvements back to the main code base, without hindering commercial deployment and improvements.

Community building is also crucial to ascertain the most important success goal for DeBinelle: An overall *improvement in code quality and reliability*. Eventually — putting aside the initial learning curve— developers will be able to easily evolve such applications without writing "redundant code", and they will spend less time on chasing down and cleaning forgotten changes and inconsistencies. The impact of DeBinelle-friendly architecture and programming styles can be quantified using standard software engineering techniques [16, 19, 26, 37].

Experimental Evaluation. Trying to plan ahead the experiments that will be eventually show the success of DeBinelle, we have to note that this database problem is also a software engineering problem. As such, we intend to go beyond experimental assessment using standard measures such as throughput, I/O, and runtime. In particular, optimizing the runtime of applying patches is not our main concern, except that we will ensure that it is en par with other development operations, and does not cause undue delays in the workflow.

Future directions. Opening this new research field introduces many questions that should be further investigated. For example, linking our approach to high-level SMOs, already introduced by the research community, should be further investigated. We could extract the higher-level SMOs from a given patch (e.g., given the patch from Figure 4,

⟨PATCH⟩ ::=@[⟨PATCHNAME⟩]@⟨HEAD⟩@@⟨BODY⟩
⟨HEAD⟩ ::={⟨METAVAR⟩;}
⟨METAVAR⟩ ::=⟨MVTYPE⟩⟨ID⟩ | ⟨MVTYPE⟩⟨ID⟩ : ⟨ISO-ID⟩
⟨BODY⟩ ::=⟨S⟩
(Statements)⟨S⟩ ::={⟨STMT⟩;}
⟨STMT⟩ ::=⟨ATOM⟩ | ⟨VARIANT⟩ | ⟨DISJUNCTION⟩ | ⟨E⟩
⟨VARIANT⟩ ::=`vIF` ⟨ID⟩ ⟨S⟩
⟨DISJUNCTION⟩ ::=`[[`⟨S⟩{`:|:`⟨S⟩}`]]`
(Ellipsis)⟨E⟩ ::=□ | ⟨E⟩ `unless` ⟨ATOM⟩

Figure 9: EBNF grammar for the patch language. ⟨Atom⟩ is a statement from the target language.

derive the semantic patch shown in Figure 6). Eventually, given the revision control system history of a project, we could alert developers for potentially missed changes, *recommend* patching strategies, and provide explanation for these (developers who have patched X have also patched Y in the past). This is closely related to the work on recommender systems in software engineering research [36]. Finally we could learn/auto-generate patches from a manual change on a single, proprietary instance of the schema file that could be then applied on other DB-variant files.

APPENDIX

A. DEBINELLE INTERNALS

We introduce the syntax for the DeBinelle patch language and discuss its implementation. We focus on two views on the task of matching patches to target code, namely as graph mappings, or as evaluating Datalog queries.

A.1 Language Definition

Figure 9 introduces our semantic patch language. The syntax leans on Coccinelle patches, but contains extensions for variant handling. At the same time, we *eliminated* some flexibility that is not required for realistic use-cases, but greatly complicates the matching process.

The general structure of the grammar can easily be mapped to previous examples. The variant conditional `vIF` has been illustrated in Figure 7. The disjunction `[[`$\mathtt{A}_1$`:|:`$\mathtt{B}_2$`:|:`$\mathtt{C}_3$`]]` indicates that *any* of $\mathtt{A}_1$, $\mathtt{B}_2$, or $\mathtt{C}_3$ can be matched. Every input isomorphism can be (internally) transformed into a disjunction. Note that, unlike logic conjunctions in target languages, the statements $\mathtt{A}_i$ are *not* restricted to Boolean-valued expressions. We use idiosyncratic operators to emphasise the difference, and to avoid symbol clashes. Ellipses (□) match *arbitrary* statements. They may be combined with a condition (`unless`), *excluding* statements in the target code. When these are encountered, the ellipsis does *not* match.

A.2 Implementation

We have outlined a number of recurring, highly non-trivial problems of database-centric application development, and devised principles to solve these. Our approach is ambitious, and we need to argue that we will be able to provide a robust implementation that scales to real-world use cases.

We have given a first overview over the DeBinelle software architecture in Section 3.3, and next provide further details.

Given a control flow graph from standard parsers, DeBinelle prepares a simplified form that contains branches, but *not* loops or jumps: Branches (`if`, `switch`, ...) entail different alternative code paths that may be treated differently by a semantic patch. Loops iterate multiple times over the *same* piece code, which is equivalent to a single path for a semantic patch. Unlike general control flow graphs (CFG) in compilers [20], CFGs are directed acyclic graphs (DAG) in our scenario: Any loop is only a *linear sequence* of statements from the DeBinelle point of view. Branches create CFG nodes with multiple outgoing edges.

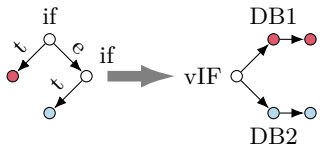


Figure 10: AST conversion into a normalised `vIF` representation.

To simplify applying the semantic patch to the target code, we normalise abstract syntax tree (AST) elements for variant branches as shown in Figure 10: Starting from a common ancestor node that represents the target language branch statement, a child graph represents each flow path, and the edge from an ancestor to a specific path is labelled with the branch condition (the illustration uses "t" for `then` and "e" for the `else` branch for the sake of simplicity).

Graph Embedding At the core, DeBinelle is concerned with mapping a semantic patch to the target language code. We deliberately employ two solution approaches to highlight different aspects of this problem: Translating a semantic patch into a Datalog program, that is matched to a given target code CFG, using standard evaluation methods, allows us to reduce graph embedding to satisfiability of Datalog programs. This is a basis for a well-defined, formally sound semantics. However, a more direct graph embedding approach that seeks a (somewhat generalised) mapping of a semantic patch into the target code CFG serves as a concrete basis for a future implementation.

The combined complexity of stratified Datalog decision problems is EXPTIME-complete. This aligns with the foundations of Coccinelle, which translates semantic patches to formulas of the likewise EXPTIME-complete computation tree logic (CTL) [10,24,44]. The benefits of using well-understood logic approaches to represent semantic patches are obvious for the provision of precise semantics. However, the enormous worst-case complexities suggest a different implementation strategy. Below, we outline an alternative approach that is less well suited to mathematical and formal reasoning, but provides substantial improvements in terms of worst-case complexity.

A.3 MSPs as Graph Mappings

The action of embedding semantic patches on target schemas and code can be understood as an embedding of the patch CFG (pCFG) into the target code CFG (tCFG).[9]

Figure 11 illustrates an embedding based on the multi-variate semantic patch and target language code shown in Figure 7. We restrict the example to two DB variants for the visual clarity. DB-variant code is indicated by differently coloured nodes (DB1 (SQLite 8–9–10); DB2 (PostgreSQL 4–5–6); the statements associated with these nodes are marked

[9]Lines leading with `+` are of no concern for embedding; they represent information that is inserted into the output.

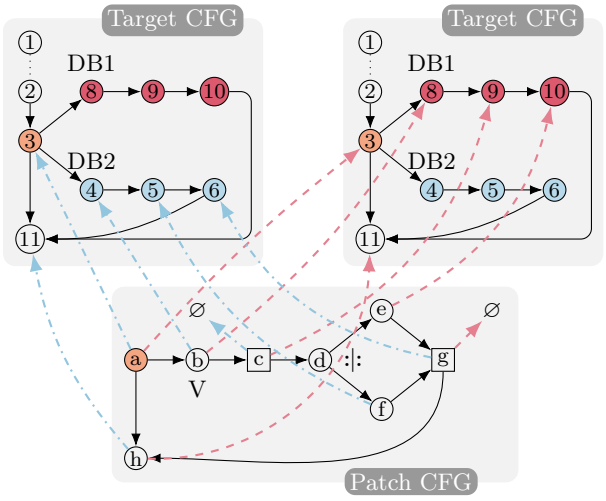


Figure 11: Semantic patch matching by graph embedding, c.f. the scenario of Figure 7. The target CFG is shown twice, to simplify the illustration.

in Figure 7). The semantic patch applies to both DB variants, and two different embeddings that connect nodes of the pCFG with (sets of) nodes of the tCFG are possible, as indicated by differently coloured/dashed mapping edges.

A pCFG node matches a tCFG node when the type of statement (conditional, function call, ...) they represent, together with any arguments, is identical, and when appropriate values for unbound meta-variables that appear in the pCFG node can be found. We compute an embedding iteratively: Given the initial node of the pCFG, the first step determines *all* matching nodes in the tCFG—we refer to these as *anchor nodes* (when a semantic patch starts with a disjunction, anchor node sets for multiple pCFG nodes must be identified, and an attempt to find a complete embedding is performed for each of them). Only ③ is an anchor node in the example tCFG, and leads to the mapping ⓐ ↦ ③.

Node ③ represents a `vIF` statement, and is therefore also a so-called *variation point*, where the control flow diverges for different DB variants; this is indicated by the background colour of node ③. Every path that starts from a variation point leads to a separate embedding, and eventually to a variant-specific patch.

Starting off the anchor node, a simultaneous traversal of patch and code DAGs matches paths by comparing the *next* pCFG and tCFG nodes (ellipses and disjunction require special treatment as detailed below). The node following ⓐ in the pCFG is ⓑ, which depends on meta-variable `V`. Two edges leave node ③ in the tCFG, leading to ⑧ and ④. Mapping ⓑ ↦ ⑧ *binds* meta-variable `V` to `DB1`, whereas mapping ⓑ ↦ ④ binds `V` to `DB2`. The process continues independently from these nodes, creating two different embeddings.

A disjunction `[[`A_1 `:|:` A_2 `:|:` A_3`]]` is represented by a dedicated node (ⓓ in Figure 11) with a child node for every choice A_i. The disjunction node itself is *not* mapped into the tCFG, but *exactly one* of the *child nodes* is.

Ellipsis nodes can map to none, one, or multiple tCFG nodes. When multiple different mappings for an ellipsis node are possible, each of them serves as starting point for a separate embedding.

A precise run-time analysis of the embedding algorithm is beyond the scope of a vision paper, so we only briefly address the computational complexity of finding an embedding.

Let $n = |V_t|$ and $m = |V_t|$ for a tCFG $G_t = (V_t, E_t)$ and a pCFG $G_p = (V_p, E_p)$. Embedding G_p into G_t is similar to the NP-complete sub-graph isomorphism (SGI) problem, except that mapping ellipses and disjunctions goes beyond a bijection. Since CFGs in our scenario do not need to contain jumps, loops or similar elements, they are planar graphs, and pattern embedding is possible in linear time in this case [17].

Ellipses and disjunctions complicate the embedding compared to planar SGI, but even a naïve method only requires polynomial time: (1) Finding all anchor nodes requires time $\mathcal{O}(n \cdot m)$—all nodes of the target graph possibly match for the initial nodes of the patch graph (in the worst case, the patch consists of a single disjunction). (2) A simultaneous traversal of pCFG and tCFG starting from an anchor node requires linear when only atomic nodes are involved. (3) Mapping an ellipsis node requires traversing the reflexive-transitive edges originating from a given initial node, which is also possible in time linear in n (computing the reflexive-transitive hull of the tCFG is naïvely possible in time $\mathcal{O}(n^3)$ [41]). (4) Mapping disjunctions is possible in linear time in the number of alternatives specified by the disjunction.

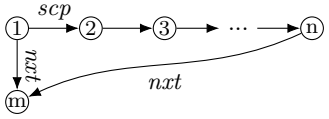
Matching ellipses with `unless` restrictions requires computing negations, which poses intricate problems in many model checking approaches [44]. However, consider the sequence ①—□ `unless X`—②. Any mapping of an *unconditional* ellipsis contains at most n nodes, and can be found in polynomial time as discussed above. We then iterate over the set of matched nodes and try to match `X` to any (sequence of) nodes. If a match is found, □ `unless X` *fails*; otherwise, the mapping for the unconditional ellipsis constitutes the mapping for □ `unless X`.[10]

Consequently, solving the decision problem for patch embedding is contained in complexity class P. This is a substantial improvement over the EXPTIME-complete CTL used in previous work [33] on whose core ideas we build.[11]

A.4 MSPs as Datalog Programs

Let us finally focus on how embedding a pCFG into a tCFG can be expressed with Datalog, which we intend to use as basis for a future formal semantics of DeBinelle. We assume familiarity with Datalog comprising recursion and (stratified) negation (Ref. [43] provides a comprehensive reference).

Encoding Target Code. Below, we encode a normalized CFG for a `create table` statement in the target code with relation *edge* : Node × Node:



In the scenario from Figure 4, the DB variant is implicitly known from the file path. We assume this information is annotated to the first node in the graph of each DDL file,

[10] Treating ellipses that are not delimited by atom nodes requires a slightly different treatment not discussed here.

[11] It should explicitly not go unnoticed that the Coccinelle tool achieves sufficiently quick practical run-times, as the tool authors have shown by measurements on real-world use-cases [33], and enjoys wide application in real-world engineering scenarios.

for instance, *variant*(**1**, **postgresql**) for the PostgreSQL-variant. Datalog predicates begin with a lower-case, and variables with a capital letter. Constants are set in bold. We encode atomic statements from the target language similarly. For instance, relation *create_table* : Node × Identifier × Bool assigns a create table statement to a node in the CFG. The table name is an identifier, and a Boolean attribute encodes whether the table was declared with the addendum `IF NOT EXISTS`. Similarly, *decl_att* : Node × Identifier × SQLType × Bool × Const encodes the declaration of an attribute at a given node, with its identifier, type, whether it is nullable, and whether and which default value is specified. Between nodes ① and ②, the edge label *scp* marks the scope of the table declaration, that is, the sequence of attributes declared. Node ⓝ represents the last attribute declared in the scope of this table. Edge label *nxt* helps navigate the CFG, and points to the next statement in the input target code. If there is no next statement, a virtual node denotes the end-of-file.

Translating Patches to Datalog. We first provide some intuition by examples. Translating the patch from Figure 5 produces the following, simple Datalog query (which we have marginally groomed for readability):

```
Q(X, T, C) :-  decl_att(X, home, T, false, C).
```

`X` matches the node in the CFG where `home` is declared, and `T` and `C` represent the meta-variables from the patch, and thereby the type and default value of attribute `home`.

Evaluating this query on the normalized CFG produces the tuple (**4**, **smallint**, **0**), as `home` is the fourth attribute declared in the given table. More generally, evaluating the query on the CFGs for the DDL code shown in Figure 4 produces three result tuples (one for each variant of `joomla.sql`). Each tuple encodes a mapping between pCFG and tCFG, and provides bindings for meta-variables.

Consider the transformed patch from Figure 7(b) top:

```
Q(V, X1, X2) :-
 variant(X0, V), rtc(X0, X1), scp(X1, X3), nxt(X1, X2),
 create_table(X1, member, _),          // Match create stmt.
 rtc(X3, X2), not p(X3, X2).           // ellipsis-unless.

p(X3, X2) :- rtc(X3, X4), decl_att(X4, expiry, T),
 edge(X4, X5), rtc(X5, X2).
```

Datalog variable `V` binds the DB-variant, `X1` binds the node where table `member` is declared, and `X2` binds to the end of this statement, and therefore, the position where the new attribute `expiry` is to be inserted. The subgoal *rtc* is the (generic) reflexive transitive closure over the *edge* relation.[12]

Algorithm Sketch. We next sketch the algorithm for translating a multivariate semantic patch to a Datalog query. Given a patch, we remove all lines prefixed with a plus, since we are interested in matching, not producing output for now. Let X, Y, Z be fresh Datalog variables. We invoke Algorithm 1 with the patch body as input

$$(D, R) \leftarrow \textsc{RecTranslate}(body, Y, Z)$$

to compute the variables D that will appear in the rule head of our Datalog query (*i.e.*, they are distinguished variables), and R as fragment of a rule body.

[12] We follow the convention where, if we are not interested in the valuation of a variable, we simply write "_". In this particular example, this symbol is introduced due to input-isomorphism *ignore-ifne* from Section 3.2.2.

Algorithm 1 Translating the body of a DeBinelle patch.

procedure RecTranslate($input, T, N$)
  $D \leftarrow \emptyset$, $R \leftarrow \epsilon$
  $X, Y \leftarrow$ FreshVariable( ) ▷ Generate new symbol
  $p \leftarrow$ FreshPredicate( ) ▷ Generate new symbol
  switch *input* **do**
    case ⟨S⟩ ▷ Translate list of statements
      if $|\langle S\rangle| = 1$ **then**
        return RecTranslate($\langle S\rangle.getFirst(), T, N$)
      $(D_1, R_1) \leftarrow$ RecTranslate($\langle S\rangle.getFirst(), T, X$)
      $(D_2, R_2) \leftarrow$ RecTranslate($\langle S\rangle.removeFirst(), X, N$)
      return $(D_1 \cup D_2, R_1 \oplus R_2)$
    case □ ▷ Translate ellipsis
      return $(\{N\}, \boxed{rtc(T, N)})$
    case ⟨E⟩ `unless` ⟨Atom⟩ ▷ Translate ellipsis-unless
      $(\_, R_A) \leftarrow$ RecTranslate($\langle$Atom$\rangle, X, Y$)
      $\boxed{p(T, N) :- \boxed{rtc(T, X)} \oplus R_A \oplus \boxed{rtc(Y, N)}.}$
      $(D, R_E) \leftarrow$ RecTranslate($\langle E\rangle, T, N$)
      return $(D, R_E \oplus \boxed{not\ p(T, N)})$
    case `[[` ⟨S1⟩ `:|:` ⟨S2⟩ `]]` ▷ Translate disjunction
      $(D_1, R_1) \leftarrow$ RecTranslate($\langle S1\rangle, T, N$)
      $(D_2, R_2) \leftarrow$ RecTranslate($\langle S2\rangle, T, N$)
      $D_p \leftarrow \{T, N, X, Y\} \cup D_1 \cup D_2$
      $p(\vec{D_p}) :- R_1 \oplus \boxed{X = T}$
      $p(\vec{D_p}) :- R_2 \oplus \boxed{Y = T}$
      return $(\{X, Y\} \cup D_1 \cup D_2, \boxed{p(\vec{D_p})})$
    case `vIF` V ⟨S⟩ ▷ Translate DeBinelle conditional
      $(D, R_S) \leftarrow$ RecTranslate($\langle S\rangle, X, Y$)
      $R \leftarrow \boxed{vIF(T), scp(T, X), variant(X, V)}$
        $\oplus \boxed{nxt(T, N)} \oplus R_S \oplus \boxed{nxt(Y, N)}$
      return (D, R)
    case ⟨Atom⟩ ▷ Translate target language statement
      return TranslateAtom(⟨Atom⟩, T, N)
  end procedure

We then add all meta-variables to D. Any Datalog variable W in D that is not bound in R is assigned a designated constant $\perp$ in the rule body (*i.e.*, $W = \perp$). This constant marks unmatched optional/alternative paths.

If the DB variant is implicit (*e.g.*, known from the file path), we further extend R by the subgoals $variant(X, V), rtc(X, Y)$ and add the meta-variable V, bound to the current DB variant, to the distinguished Datalog variables D. Finally, ordering the distinguished variables (by some order), we obtain the (safe) Datalog query $Q(\vec{D}) :- R$.

The idea in Algorithm 1 is to recursively assemble a Datalog rule body, given two arguments T, N. Variable T (for "this") represents the current node matched in the normalized target code CFG, and N represents the next patch statement. We use boxes to mark rule body fragments, so that it is easier to distinguish them from pseudo-code. Operator $\oplus$ simply concatenates rule body fragments.

We do not show details regarding the implementation of procedure TranslateAtom, which depends on the input target language (*e.g.*, SQL, PHP, or C++). Note that TranslateAtom will not call RecTranslate. In addition, while the translations shown before have been marginally groomed for presentation (having removed redundant subgoals that are merely artefacts of the translation), they can actually be generated in this fashion.

2. REFERENCES

[1] F. Afrati, T. Andronikos, V. Pavlaki, E. Foustoucos, and I. Guessarian. From CTL to Datalog. In *Proceedings of the Paris C. Kanellakis Memorial Workshop on Principles of Computing & Knowledge*, PCK50, page 72–85, 2003.

[2] S. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley Publishing, 1st edition, 2003.

[3] S. W. Ambler and P. J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[4] C. Athinaiou and H. Kondylakis. VESEL: VisuaL Exploration of Schema Evolution using Provenance Queries. In *EDBT Workshops*, 2019.

[5] J. Bacon. *The art of community: Building the new age of participation*. O'Reilly Media, Inc., 2012.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[7] P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *ACM SIGMOD*, pages 1–12, 2007.

[8] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller. A foundation for flow-based program matching: Using temporal logic and model checking. In *ACM SIGPLAN-SIGACT*, POPL '09, pages 114–126, New York, NY, USA, 2009. ACM.

[9] R. Cilibrasi and P. M. Vitanyi. Clustering by compression. *IEEE Trans. Inf. Theor.*, 51(4), Apr. 2005.

[10] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[11] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. H. Weber. Understanding database schema evolution: A case study. *Sci. Comput. Program.*, 97:113–121, 2015.

[12] A. Cleve and J.-L. Hainaut. *Co-transformations in Database Applications Evolution*, pages 409–421. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[13] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. *VLDB J.*, 22(1):73–98, 2013.

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

[15] C. A. Curino, L. Tanca, H. J. Moon, and C. Zaniolo. Schema evolution in Wikipedia: Toward a Web Information System Benchmark. In *Proc. ICEIS'08*, 2008.

[16] T. DeMarco. *Controlling software projects: management, measurement & estimation*, volume 1133. Yourdon Press New York, 1982.

[17] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Graph Algorithms and Applications I*, pages 283–309. World Scientific, 2002.

[18] B. Fong and D. I. Spivak. Seven Sketches in Compositionality: An Invitation to Applied Category Theory, 2018. cite arxiv:1803.05316Comment: 243 pages.

[19] X. Franch and J. P. Carvallo. Using quality models in software package selection. *IEEE software*, 20(1):34–41, 2003.

[20] D. Grune, K. van Reeuwijk, H. Bal, C. Jacobs, and K. Langendoen. *Modern Compiler Design*. Springer New York, 2012.

[21] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, and W. Lehner. Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In *ACM SIGMOD*, pages 1101–1116, 2017.

[22] K. Herrmann, H. Voigt, T. B. Pedersen, and W. Lehner. Multi-schema-version data management: data independence in the twenty-first century. *VLDB J.*, 27(4):547–571, 2018.

[23] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *ACM SIGPLAN-SIGACT POPL*, pages 371–384, 2011.

[24] M. Huth and M. Ryan. *Logic in computer science - modelling and reasoning about systems*. Cambridge University Press, 01 2000.

[25] M. Joblin, S. Apel, and W. Mauerer. Evolutionary trends of developer coordination: a network approach. *Empirical Software Engineering*, 22(4):2050–2094, 2017.

[26] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle. From developer networks to verified communities: A fine-grained approach. In *ICSE*, ICSE '15, pages 563–573, 2015.

[27] H. J. Kang, F. Thung, J. Lawall, G. Muller, L. Jiang, and D. Lo. Semantic Patches for Java Program Transformation (Experience Report). In A. F. Donaldson, editor, *ECOOP 2019*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:27, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[28] J. Lawall and G. Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 601–614, Boston, MA, July 2018. USENIX Association.

[29] D.-Y. Lin and I. Neamtiu. Collateral Evolution of Applications and Databases. In *Proc. IWPSE-Evol'09*, 2009.

[30] W. Mauerer and M. C. Jaeger. Open source engineering processes / open source-entwicklungsprozesse. *it - Information Technology*, 55(5):196–203, 2013.

[31] P. Montero and J. A. Vilar. TSclust: An R package for time series clustering. *Journal of Statistical Software*, 62(1):1–43, 2014.

[32] H. J. Moon, C. Curino, M. Ham, and C. Zaniolo. PRIMA: Archiving and querying historical data with evolving schemas. In *ACM SIGMOD*, pages 1019–1022, 2009.

[33] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *ACM SIGOPS/EuroSys*, pages 247–260, New York, NY, USA, 2008. ACM.

[34] D. Qiu, B. Li, and Z. Su. An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications. In *Proc. ESEC/FSE'13*, 2013.

[35] R. Ramsauer, D. Lohmann, and W. Mauerer. The List is the Process: Reliable Pre-Integration Tracking of

Commits on Mailing Lists. In *ICSE*, ICSE '19, page 807–818. IEEE Press, 2019.

[36] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[37] J. M. Roche. Software metrics and measurement principles. *ACM SIGSOFT Software Engineering Notes*, 19(1):77–85, 1994.

[38] J. F. Roddick. Schema Evolution in Database Systems: An Annotated Bibliography. *SIGMOD Rec.*, 21(4):35–40, Dec. 1992.

[39] M. Schaefer and O. de Moor. Specifying and implementing refactorings. In *ACM OOPSLA*, pages 286–301, New York, NY, USA, 2010. ACM.

[40] R. E. Schuler and C. Kessleman. A High-level User-oriented Framework for Database Evolution. In *Proc. SSDBM'19*, pages 157–168, 2019.

[41] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[42] M. Stonebraker. My Top Ten Fears about the DBMS Field. In *IEEE ICDE*, pages 24–28, 2018.

[43] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, Inc., USA, 1988.

[44] M. Y. Vardi. Model checking for database theoreticians. In *ICDT*, pages 1–16, Berlin, Heidelberg, 2005. Springer-Verlag.

[45] P. Vassiliadis, M.-R. Kolozoff, M. Zerva, and A. V. Zarras. Schema evolution and foreign keys: A study on usage, heartbeat of change and relationship of foreign keys to table activity. *Computing*, 101(10):1431–1456, Oct 2019.

[46] M. Verbaere, R. Ettinger, and O. de Moor. Jungl: A scripting language for refactoring. In *ICSE*, ICSE '06, pages 172–181, New York, NY, USA, 2006. ACM.

[47] B. Wall and R. A. Angryk. Minimal data sets vs. synchronized data copies in a schema and data versioning system. In *Workshop for Ph.D. students in information & knowledge management, IPKM*, pages 67–74, 2011.

[48] Y. Wang, J. Dong, R. Shah, and I. Dillig. Synthesizing Database Programs for Schema Refactoring. In *Proc. PLDI 2019*, pages 286–300, 2019.

[49] D. A. Watt. *Programming Language Design Concepts.* John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

[50] S. Wu and I. Neamtiu. Schema evolution analysis for embedded databases. In *IEEE ICDE Workshops*, pages 151–156, April 2011.