DeBinelle: Semantic Patches for Coupled Database-Application Evolution

Stefanie Scherzinger*
University Passau
Germany
stefanie.scherzinger@uni-passau.de

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

Haridimos Kondylakis
FORTH-ICS
Heraklion, Greece
kondylak@ics.forth.gr

* Authors 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 enforce *collateral evolution*, which is a recognised, important research problem. In this demonstration, we show a new dimension to this problem, in software that supports alternative database backends: 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 for software developers. We introduce *DeBinelle*, a novel framework and domain-specific language for *semantic patches* that abstracts DB-variant schema changes and coupled program code into a single, unified representation. DeBinelle further offers a novel alternative to manually evolving coupled schemas and code. DeBinelle considerably extends established, seminal results in software engineering research, supporting several programming languages, and the many dialects of SQL. It effectively eliminates the need to perform vendor-specific changes, replacing them with intuitive semantic patches. Our demo of DeBinelle is based on real-world use cases from reference systems for schema evolution.**

***Index Terms*—Databases; Evolution; Semantic Patches**

I. Introduction

Decades of research have contributed frameworks for managing database schema evolution. Early proposals date back to the 80s [1], among the more recent are PRISM [2], InVerDa [3], or CHiSEL [4]. Yet from the perspective of software developers, the schema does not evolve in isolation. When the database schema changes, other parts of the code, such as GUIs and application logic, commonly change as well. This coupling causes *collateral evolution*, a known and important challenge [5]

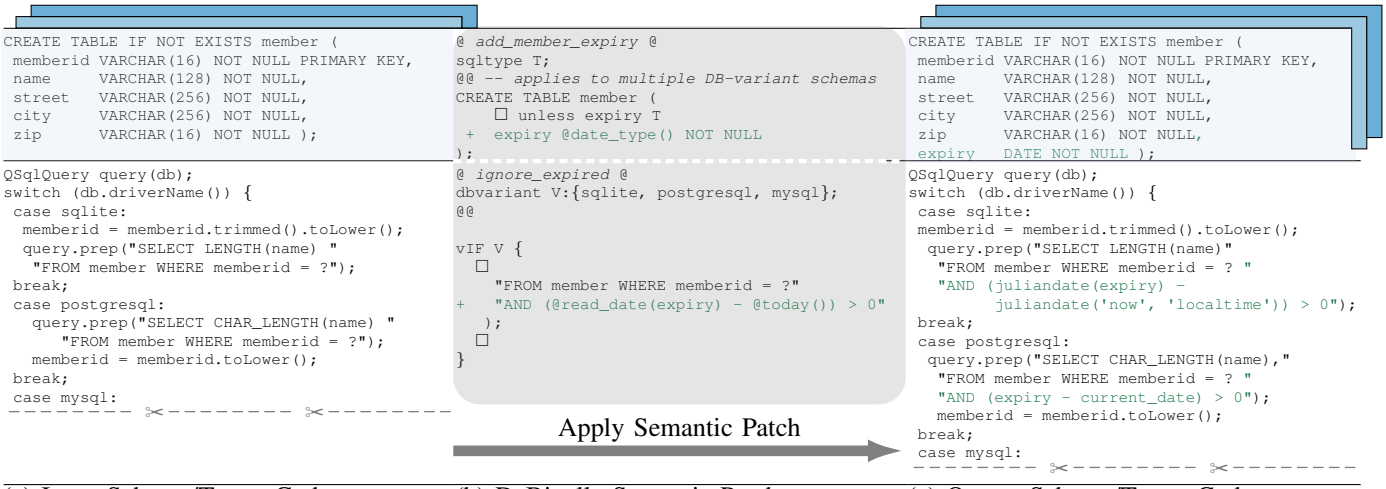
Numerous applications support several alternative database backends (or *variants*), which adds complexity [6]: There are widely adopted database applications where SQL dialects[1] lead to multiple, essentially identical schemas to be declared and maintained. This does not affect only legacy systems (whose use will eventually cease), but also "new generation" applications, such as MediaWiki, the software behind Wikipedia.

When such applications are installed, the operations team chooses one of the supported database variants. Yet during application development, each schema variant is again coupled with code. Despite the fact that the evolution histories of prominent database applications, for instance, the well-known content management systems MediaWiki and Joomla!, have been extensively studied as reference systems for schema evolution [7]–[11], only a few studies consider collateral evolution [5], [12], [13], and no studies known to us target the added challenge of maintaining vendor-variant database schemas in parallel. Moreover, relational mappings do not resolve the problem—they merely introduce one more software layer [14]. To the best of our knowledge, all existing tools supporting database schema evolution focus on *single-schema* evolution.

Contributions: In this demonstration, we unfold the problem of collateral evolution, where database and application evolution are coupled. This involves the need for performing recurring, conceptually identical changes to database schema and program code, for each *database variant*. While this is a real and important problem, it has so far been overlooked in research. To the best of our knowledge, ours is the first paper addressing this specific problem. Then, we present the DeBinelle framework, implemented in OCaml and based on the ideas of Coccinelle [15], deliberately delivered as a command line tool. It can be readily integrated into *any* development process, with minimal upfront costs. Further, we introduce and explain *DeBinelle semantic patches* that capture the changes to the vendor-specific database schemas, as well as the application code, on a conceptual level. We demonstrate the usefulness of our approach and allow demo participants to issue semantic patches for real world scenarios, effectively applying patches with only a few lines, to hundreds lines of code. We further provide insights into DeBinelle internals. At its heart, DeBinelle computes a graph embedding, mapping the control flow graph of a semantic patch onto the control flow graph of the target code. In the demo, we also show how a Datalog engine may compute these mappings.

Outline: In Section II, we present the DeBinelle framework along with its architecture, and the concept of semantic patches. In Section III we present related work, showing that DeBinelle fills an important gap in existing research. Section IV presents the demonstration scenario and concludes this paper.

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



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

Fig. 1. A scenario based on real-world patterns observed in large projects. Above: Attribute `expiry` is added to relation `member`. Below: Queries are also adapted. In both cases, one of *many* possible matching code snippets is shown on the left. The semantic patch comprising two parts (middle) produces the output on the right for the given target code (left). The scissors mark code not shown.

II. The DeBinelle Framework

We walk through a DeBinelle use case, before we outline the DeBinelle architecture and workflow.

A complete example: In Figure 1, we show an example from a hypothetical, BiblioteQ-like library management software. This example unifies several patterns that we have repeatedly observed in real-world projects.

We distinguish three DB variants, to keep the example manageable. The upper half of Figure 1 shows the addition of attribute `expiry` to the relation managing memberships. To the left, we see an excerpt from *one* data definition language (DDL) file, out of many variants (here, PostgreSQL). Top centre, we see the patch appending the new attribute to the table declaration. The syntax resembles that of syntactic patches: The rule header (line 1) declares the patch name, surrounded by the markers `@...@`. The header declares *meta-variables*, where `T` matches some SQL type. Starting with line 4, we specify the transformations. A minus symbol in front of a line states that matching lines will be removed. Lines leading with a plus are added. The ellipsis operator □ combined with `unless` ensures that the patch only matches table declarations where this attribute does *not* (yet) exist (for some SQL type `T`). This prevents redundant introduction.

Note that the patch can be applied to the input, even though line 4 in the patch, and line 1 in the table declaration, differ in the restriction `IF NOT EXISTS`, as DeBinelle exploits so-called *input isomorphisms*. These are sets of semantically equivalent keywords that can be matched against a given piece of code. As in [16], DeBinelle input isomorphisms are compact and—in isolation—easy to write and read (seemingly trivial even), but effective when collectively applied to the target code. DeBinelle ships with a customisable library of such isomorphisms. For example, the following output isomorphism *`date_type`* inserts the correct type, based on the DB-variant known from the context (here, the file path):

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

Continuing our example in Figure 1, 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. To the top right, we show the modified target code.

This schema change is coupled with code changes: In the original target code shown bottom left, a SQL query retrieves the length of library member names. This query is assembled from strings. In line 2, the control flow diverges: Switch labels distinguish between database variants.

Bottom centre, we show the DeBinelle patch. 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. (DeBinelle is aware of SQL syntax and semantics, and automatically inserts a single space at the end of the where-condition.)

Line 2 of the semantic patch introduces variant anchors, for SQLite, PostgreSQL, and MySQL. 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`* allows for matching `LENGTH(name)` from the patch against both lines 5 *and* 9 in the input target code, even though they differ syntactically. This is ensured by the following input-isomorphism.

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

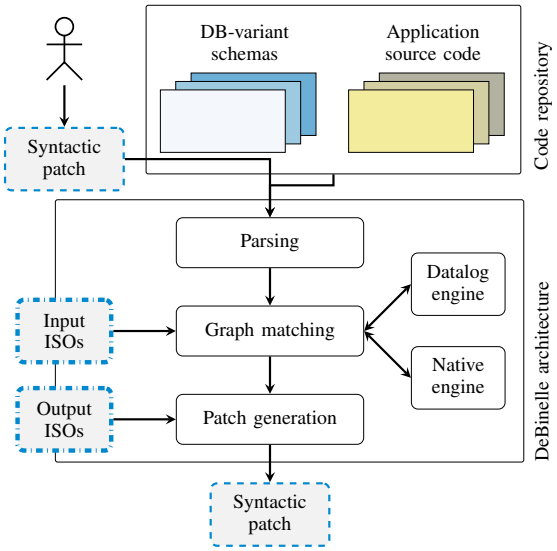


Fig. 2. DeBinelle system architecture and data flow.

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` handle this, but are not shown, due to lack of space. Line 8 thus computes the membership expiration date.

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 meta-variable `V`. When matched in the target code, DeBinelle can infer the database 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). To the bottom right, we show the output target code.

A real-world example: We next discuss a more complex example that is modelled after a real use case. The short semantic patch below converts the combination of `CREATE TABLE` and `UNIQUE INDEX` to a primary key constraint, effectively joining on the table name. Using manual changes, the corresponding action in project MediaWiki (modelled after commit `267d99f`) requires inspecting over 200 `CREATE TABLE` statements. The output isomorphism `@strconcat` ensures the constraints are prefixed by "`PK`".

```
1  @@ sqlid T, A1, A2, I; @@
2  CREATE TABLE T (
3  □
4  + CONSTRAINT @strconcat('PK', T) PRIMARY KEY(A1, A2)
5  );
6  □
7  - CREATE UNIQUE INDEX I ON T (A1, A2);
```

System architecture: The architecture of our framework is shown in Figure 2. Structurally, DeBinelle is a variant-aware, multi-language source-to-source transpiler: The input are semantic patches, as written by the user of DeBinelle, and a set of input files in the target languages. This includes SQL-DDL statements in several database variants, as well as the application source code, contained within the code repository. Following usual patterns in compiler construction, DeBinelle parses the input files into intermediate program representations, based on control flow graphs (CFGs).

The patch is also parsed into a graph. Input isomorphisms are expanded on the CFG of the DeBinelle semantic patch. Then, the patch CFG is embedded into the graph target code CFG. This embedding can be either computed by leveraging a Datalog engine, or a native implementation. The matching detects *variation points* that require different treatment depending on the database variant. The final phase performs any replacements (additions and/or deletions)[2] mandated by the DeBinelle semantic patch and applies output isomorphisms to generate different syntactic changes for each relevant variant.

III. Related Work

Schema Evolution Tool Support. For practitioners, Flyway, Liquibase, Rails Migrations, and DBmaestro Teamwork enable systematic migration between schema versions, supporting most changes in data definition and modification language (DDL & DML) files. Most of them generate small programs to perform the evolution at the schema level. Unfortunately, none of these tools support collateral evolution. From research, there have been various contributions for database schema evolution for a *single* schema variant. For example, model management [17] handles multiple schema versions by allowing to match, diff, and merge existing schemas to derive mappings between these schemas. PRIMA [18], PRISM & PRISM++ [2], CHiSEL [4], VESEL [19], ScaDaVer [20], and InVerDA [3] exploit simple and intuitive *schema modification operations* (SMOs) for describing schema evolution, and most of them enable schema and data migration, as well as automated query rewriting between the versions. MIGRATOR [21] automatically updates SQL statements upon the schema changes. Symmetric Lenses [22] facilitates read and write access along a bidirectional mapping, while auxiliary tables persist the complements to not lose any data. With DeBinelle, we can rewrite the *multiple* DB-variant specifications of integrity constraints, default values, or types and stored procedures. Most importantly, DeBinelle is the first tool, to our knowledge, that even targets collateral evolution.

Software Evolution Tool Support. Eliminating manual labour and recurring, repetitive tasks by automation is intensively studied in contemporary software engineering research. For example, the Coccinelle [15] 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 in later versions for subsets of C++ and Java), but lacks support for generating variant output, which DeBinelle is capable of. Although numerous research works focus on refactoring and re-engineering programs, databases are usually only mentioned as (yet) another component that

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

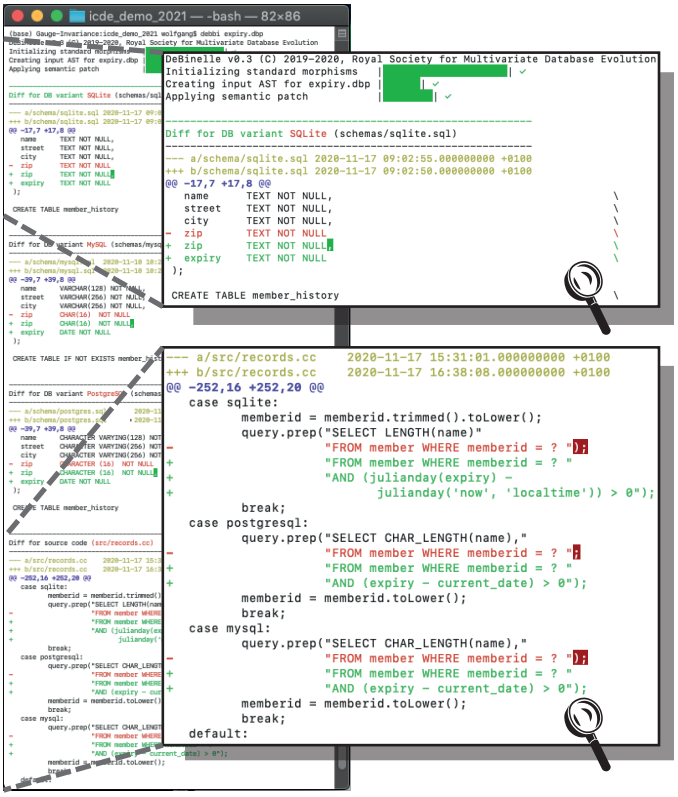


Fig. 3. A screenshot of DeBinelle in action. Instead of providing integration with one specific IDE, a generic command line tool turns a DeBinelle semantic patch into a series of syntactic patches (as shown in the tool output) for schemas and code, which can be integrated into any revision control mechanism.

is addressed via an abstraction layer, thus disregarding the problems addressed by DeBinelle.

IV. Demonstration Scenario

Our demonstration shows the benefits of using DeBinelle in real-world evolution scenarios. A screenshot of the system is shown in Figure 3:

Real-World Examples: We will present real-world evolution scenarios from well-known database applications. We will show, from github logs scenarios, that the application code, along with the database schema, evolve. We will point out the huge maintenance effort, presenting the total lines of code that require a manual effort, also highlighting limitations of existing tools.

Introduction to DeBinelle Semantic Patches: Then we will begin with small examples that will familiarise conference participants with the DeBinelle notation, showing the capabilities of DeBinelle semantic patches. Input and output isomorphisms will be explained, revealing that adding yet another database variant introduces just additional isomorphisms.

We will further disclose DeBinelle internals, such as the Datalog programs computing the mapping between a semantic patch and the control flow graph obtained from the sources, focusing on our novel ideas here.

Semantic Patches for Real-World Examples: We will show that a few lines of multivariate semantic patches are adequate to support the evolution of both the code and the database schema, contrasting the effort in terms of lines of code required for both approaches.

Mini-Game: We will challenge conference participants to implement specific evolution scenarios, trying to estimate the lines of code required for the corresponding semantic patch.

Acknowledgments: This project was partly supported by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) – 385808805, and the BOUNCE H2020 EU project (GA #777167). We thank Edson Lucas for drawing Figure 2.

References

[1] J. F. Roddick, "Schema Evolution in Database Systems: An Annotated Bibliography," *SIGMOD Rec.*, vol. 21, no. 4, pp. 35–40, Dec. 1992.
[2] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," *VLDB J.*, vol. 22, no. 1, pp. 73–98, 2013.
[3] K. Herrmann, H. Voigt, T. B. Pedersen, and W. Lehner, "Multi-schema-version data management: data independence in the twenty-first century," *VLDB J.*, vol. 27, no. 4, pp. 547–571, 2018.
[4] R. E. Schuler and C. Kessleman, "A High-level User-oriented Framework for Database Evolution," in *Proc. SSDBM'19*, 2019, pp. 157–168.
[5] D. Qiu, B. Li, and Z. Su, "An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications," in *ESEC/FSE*, 2013.
[6] P. Vassiliadis, "Profiles of Schema Evolution in Free Open Source Software Projects," in *Proc. ICDE*, 2021.
[7] 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*, vol. 4, no. 2, pp. 117–128, 2010.
[8] C. A. Curino, L. Tanca, H. J. Moon, and C. Zaniolo, "Schema evolution in Wikipedia: Toward a Web Information System Benchmark," in *Proc. ICEIS*, 2008.
[9] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. H. Weber, "Understanding database schema evolution: A case study," *Sci. Comput. Program.*, vol. 97, pp. 113–121, 2015.
[10] 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*, vol. 101, no. 10, pp. 1431–1456, 2019.
[11] D. Brainerd, W. Mauerer, and S. Scherzinger, "Replicability and Reproducibility of a Schema Evolution Study in Embedded Databases," in *Proc. ER Workshops*, 2020, pp. 210–219.
[12] D.-Y. Lin and I. Neamtiu, "Collateral Evolution of Applications and Databases," in *Proc. IWPSE-Evol'09*, 2009, p. 31–40.
[13] A. Cleve and J.-L. Hainaut, "Co-transformations in Database Applications Evolution," in *Proc. GTTSE*, 2006.
[14] A. Jaimoon and T. Suwannasart, "Impact Analysis of Database Schema Changes on Hibernate Source Code and Test Cases," in *ICSEB*, 2019.
[15] 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 *Proc. SIGPLAN-SIGACT*, 2009.
[16] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and Automating Collateral Evolutions in Linux Device Drivers," in *Proc. SIGOPS/EuroSys*, 2008.
[17] P. A. Bernstein and S. Melnik, "Model management 2.0: Manipulating richer mappings," in *Proc. SIGMOD*, 2007, pp. 1–12.
[18] H. J. Moon, C. Curino, M. Ham, and C. Zaniolo, "PRIMA: Archiving and querying historical data with evolving schemas," in *SIGMOD*, 2009.
[19] C. Athinaiou and H. Kondylakis, "VESEL: VisuaL Exploration of Schema Evolution using Provenance Queries," in *EDBT Workshops*, 2019.
[20] B. Wall and R. A. Angryk, "Minimal data sets vs. synchronized data copies in a schema and data versioning system," in *Proc. IPKM*, 2011.
[21] Y. Wang, J. Dong, R. Shah, and I. Dillig, "Synthesizing Database Programs for Schema Refactoring," in *Proc. PLDI*, 2019, pp. 286–300.
[22] M. Hofmann, B. C. Pierce, and D. Wagner, "Symmetric lenses," in *Proc. POPL*, 2011, pp. 371–384.