A Port Graph Rewriting Approach to Relational Database Modelling

Maribel Fernández, Bruno Pinaud, Janos Varga

A Port Graph Rewriting Approach to Relational Database Modelling

Maribel Fernández[1], Bruno Pinaud[2] and János Varga[1]

[1] King's College London, Department of Informatics, London WC2B 4BG
[2] LaBRI, Université de Bordeaux, 351 Cours de la Libration, 33405 Talence cedex
janos.varga@kcl.ac.uk

Abstract. We present new algorithms to compute the Syntactic Closure and the Minimal Cover of a set of functional dependencies, using strategic port graph rewriting. We specify a Visual Domain Specific Language to model relational database schemata as port graphs, and provide an extension to port graph rewriting rules. Using these rules we implement strategies to compute a syntactic closure, analyse it and find minimal covers, essential for schema normalisation. The graph program provides a visual description of the computation steps coupled with analysis features not available in other approaches. We prove soundness and completeness of the computed closure. This methodology is implemented in PORGY.

Keywords: relational databases, database design, port graph, graph transformation, functional dependency, minimal cover

1 Introduction

Relational database design includes conceptual and logical modelling, as well as physical modelling. The theory behind these steps is well-understood (it is part of the syllabus of many databases courses [16]), and highlights the advantages of developing normalised database designs. Yet, database professionals often consider normalisation too cumbersome and do not apply normalisation theory, due to the lack of adequate tools to support logical modelling [6].

Formal, graph-based approaches to database design have used labelled graphs or hypergraphs [1,5,7]. We advocate a new approach to database modelling using *attributed port graphs*, which are graphs where edges are connected to nodes at specific points, called ports. Attributes of nodes, edges and ports are used to represent properties of the system modelled. Port graphs were introduced in [2] to model biochemical systems and have been used in various domains [14]. Port graphs are a good data structure to store and to visualise relational schema: ports provide additional visual information about the design. We propose to represent relational attributes and functional dependencies as nodes, and use edges to link attributes and dependencies; ports indicate the role of the attribute in the dependency. This representation has advantages when computing properties of the schema, such as syntactic closure of the set of dependencies, a crucial step in producing a normalised schema. We specify an algorithm to compute closures

using *port graph rewriting rules* controlled by strategies. Our system has been implemented in PORGY [3] – a visual, interactive modelling tool. PORGY provides a graphical interface to specify an initial model, port graph rewriting rules and strategies. It displays the set of rewrite derivations (a *derivation tree*) and includes features such as cycle detection, to facilitate debugging.

Summarising, our main contributions are:

1. a Visual Domain Specific Language (VDSL) specifically tailored to model relational database schemata (Section 3);
2. a new visual representation of Armstrong's axioms to infer functional dependencies, using the port graph VDSL mentioned above (Section 4.1);
3. a sound and complete strategic graph program to compute the syntactic closure of a set of functional dependencies, with examples (Sections 4.2,4.3);
4. an implementation[3] in PORGY, together with a set of techniques to query the relational database design, using PORGY's derivation tree and graphical interface to analyse properties of the model: In particular, we show how to solve *the membership problem* (Section 4.4);
5. a strategy and a set of transformation rules to simplify sets of dependencies, as required to compute a minimal cover (Section 5).

Related Work. Hypergraphs are used for relational database design in [7,13]. Using directed graphs candidate keys of a relation are computed in polynomial time [24]. A special family of labelled graphs, FD-graphs, were introduced in [5] to obtain closures of functional dependencies. In terms of graph transformations for database modelling we highlight two works. Hypergraph rewriting was used for the representation of functional dependencies [7] and Triple Graph Grammars were used to optimise a database schema [17].

Our contribution and main difference with respect to these works is the design of a domain-specific visual language with emphasis on interactive modelling, including strategies to control the application of rules, and the use of the derivation tree as part of the visualisation framework, giving the modeller access to all the sequences of transformation steps, to facilitate the analysis of the system. Port graphs were used to compute transitive closures in [27]. Here we compute the full Armstrong closure (not just transitive closure), and show how to use the derivation tree to analyse closures and answer queries about the database model, such as whether a given functional dependency is in the closure of a set of dependencies (the membership problem), and compute minimal covers.

PORGY's strategy language is strongly inspired by PROGRES [25], GP [23] and by strategy languages developed for term rewriting [11,19]. None of the available graph rewriting tools permits users to visualise the derivation tree, as in PORGY, where users can interactively visualise alternative derivations, follow the development of specific redexes, etc. When computing the closure of a set of dependencies, the derivation tree permits to see how each dependency is generated, offering a direct visualisation of the inference steps according to Armstrong's axioms.

[3] github.com/janos-varga/Porgy

2 Background

2.1 Relational Databases

We assume that the reader is familiar with the theory of logical design of relational databases [22], in particular, the definitions of: *relation schema*, *attribute*, *candidate key* and *functional dependency* (FD). We refer to a single attribute with letters from the beginning of the alphabet $A, B, \ldots$ and to attribute sets with letters from the end of the alphabet W, X, Y, Z. Let $\mathcal{R}(\mathcal{A}) = \{R_1, \ldots, R_k\}$ be a set of relation schemata over a set $\mathcal{A}$ of attributes. Let $\mathcal{FD} = \{\Sigma_1, \ldots, \Sigma_k\}$ be the respective sets of functional dependencies and $\mathcal{CK} = \{C_1, \ldots, C_k\}$ be the respective sets of candidate keys. A relational database schema is a tuple $\mathcal{DB} = (\mathcal{R}(\mathcal{A}), \mathcal{FD}, \mathcal{CK})$.

We assume familiarity with the inference rules known as Armstrong's Axioms: Reflexivity (or Trivial Dependency), Augmentation, Transitivity, Union, Decomposition, Pseudotransitivity. These rules are sound and complete [4,9]. From now on, by syntactic closure we will mean Amstrong's syntactic closure, that is, the set Σ^+ of all FDs that can be inferred from Σ using Armstrong's Axioms, or equivalently, using the sound and complete subset consisting of Reflexivity, Transitivity and Augmentation, stated below following Beeri et al. [9].

(A1) Reflexivity: **if** $Y \subseteq X$ **then** $X \to Y$.
(A2) Augmentation: **if** $Z \subseteq W$ and $X \to Y$ **then** $XW \to YZ$.
(A3) Transitivity: **if** $X \to Y$ and $Y \to Z$ **then** $X \to Z$.

Syntactic closures are used to compute **minimal covers** of sets of FDs. The *minimal cover* Σ_{min} of Σ is a set of dependencies that fully represent Σ and satisfy the following three conditions [21]:

1. all the FDs in Σ_{min} have singleton right sides;
2. Σ_{min} is left-reduced: if one attribute is removed from any left side then Σ can no longer be inferred from Σ_{min};
3. Σ_{min} is nonredundant: if any FD is removed from Σ_{min} then Σ can no longer be inferred from it.

Our goal is to provide visual algorithms to compute syntactic closures and minimal covers. This work assumes that a) FDs have singleton right sides and b) there are no cyclical dependencies. Assumption (a) is standard in the relational database literature, without loss of generality, under Armstrong's Decomposition rule. The problem of cyclical dependencies reduces to kernel search in a directed graph which is NP-complete.

2.2 Port Graph Rewriting and Porgy

We recall the notion of attributed port graph rewriting (see [14] for more details).

Definition 1 (Attributed port graph). *An* attributed port graph $G = (V, P, E, D)_{\mathcal{F}}$ *is a tuple* (V, P, E, D) *of pairwise disjoint sets where:*

- *V is a finite set of nodes; $n, n_1, \ldots$ range over nodes;*
- *P is a finite set of ports; $p, p_1, \ldots$ range over ports;*
- *E is a finite set of edges between ports; $e, e_1, \ldots$ range over edges; two ports may be connected by more than one edge;*
- *D is a set of records, which are sets of pairs attribute-value;*

and a set $\mathcal{F}$ of functions Connect, Attach and $\mathcal{L}$abel such that:

- *for each edge $e \in E$, Connect(e) is the pair (p_1, p_2) of ports connected by e;*
- *for each port $p \in P$, Attach(p) is the node n to which the port belongs;*
- *$\mathcal{L}abel : V \cup P \cup E \mapsto D$ is a labelling function that returns a record for each element in $V \cup P \cup E$.*

For each node $n \in V$, $\mathcal{L}abel(n)$ contains an attribute Interface whose value is the list of names of its ports.

A *port graph rewrite rule* is itself a port graph $L \Rightarrow_C R$ consisting of two subgraphs L and R, called *left-hand side* and *right-hand side*, respectively, together with an *arrow* node that links them. Each rule is characterised by its arrow node, which has a unique name (the rule's label), an optional attribute *Where* defining a Boolean condition C that restricts the rule's matching, and ports to control the rewiring operations when rewriting steps are computed. Each port in the arrow node has an attribute *Type* that can have one of three different values: *bridge*, *wire* and *blackhole*. A port of type *bridge* must have edges connecting it to L and to R (one edge to L and one or more to R): it thus connects a port from L to ports in R. A port of type *blackhole* must have edges connecting it only to L (one edge or more). A port of type *wire* must have exactly two edges connecting to L and no edge connecting to R.

The ports and edges associated with the arrow node specify a mapping between ports in the left and right-hand sides of the rule, following the Single-PushOut approach [20]. This mapping is used during rewriting, to redirect the edges that connect the redex to the rest of the graph once the redex is rewritten (as explained below).

For examples of rewrite rules, we refer the reader to Section 4. It is possible to specify a rule condition requiring that a particular edge does NOT exist in the graph to be rewritten. In PORGY such conditions are graphically represented as a double line grey edge with an X, which is called an anti-edge [15].

A *match* $g(L)$ of the left-hand side is found in G if there is a total port graph morphism g from L to G such that if the arrow node has an attribute *Where* with value C, then $g(C)$ is true in G. C is of the form *saturated*$(p_1) \wedge \ldots \wedge$ *saturated*$(p_n) \wedge B$, and *saturated*$(g(p_i))$ holds if there are no edges between $g(p_i)$ and ports outside $g(L)$ in G – this ensures that no edges will be left dangling in rewriting steps. B is a Boolean expression such that all its variables occur in L.

Let G be a port graph. A *rewrite step* $G \Rightarrow H$ via the port graph rewrite rule $L \Rightarrow_C R$ is obtained by replacing in G a subgraph $g(L)$ by $g(R)$, where g is a morphism from L to G satisfying C, and connecting $g(R)$ to the rest of the graph as indicated by the arrow-node edges in the rule: Any edges arriving

to a port in g(L) connected by a bridge arrow port to R are transferred to the corresponding ports in $g(R)$; edges connecting to ports in $g(L)$ that are connected to a blackhole port in the arrow node are deleted. Wire ports in the arrow node trigger a rewiring: the ports in G that connect to the ports in $g(L)$ associated to a wire port in the arrow node are linked by an edge in the rewritten graph.

A sequence of rewriting steps is called a *derivation*. A *derivation tree* is a collection of rewriting derivations with a common root.

PORGY [14] includes functionality to create port graphs and port graph rewrite rules, and to apply rules to graphs according to user-defined strategies. The functions *Connect* and *Attach* (see Definition 1) are represented as attributes in records (i.e., records contain data attributes, visualisation attributes such as colour or shape, and structural attributes such as Connect and Attach). Rules are displayed as graphs, and edges that run between ports of L, R and the arrow node are coloured red to distinguish them from normal edges. PORGY also provides a visual representation of the rewriting derivations, which can be used to analyse the rewriting system. PORGY's *strategy language* allows us to control the way derivations are generated. We can specify not only the rule to be used in a rewriting step, but also the position where the rule should (or should not) be applied. Formally, the rewriting engine works with *graph programs*.

Definition 2 (Graph Program). *A graph program consists of a* located port graph*, a set of port graph rewriting rules, and a strategy expression. A* located port graph *is a port graph with two distinguished subgraphs: a* position *subgraph and a* banned *subgraph, denoted* G^P_Q*. Rewrite rules can only be applied to G if they match a subgraph which superposes P and does not superpose Q.*

We briefly describe below the strategy constructs that we use in our programs (see [14] for more details). The keywords `crtGraph`, `crtPos`, `crtBan` denote, respectively, the current graph being rewritten and its Position and Banned subgraphs. For example, the strategy expression `setPos(crtGraph)` sets the position graph as the full current graph. If T is a rule, then the strategy `one`(T) randomly selects one possible redex for rule T in the current graph G, which should superpose the position subgraph P and not overlap the banned subgraph Q. This strategy fails if the rule cannot be applied. Constants id and fail denote success and failure, respectively. `while`(S)[(n)]`do`(S') executes strategy S' (not exceeding n iterations if the optional parameter n is specified) while S succeeds. `repeat`$(S)[max\ n]$ repeatedly executes a strategy S, not exceeding n times. It can never fail (when S fails, it returns id).

3 Port Graphs for Database Modelling

First, we define a visual domain specific language (VDSL) for logical design of relational databases. It includes a class of attributed port graphs to represent objects of a relational database, and a language to specify rewrite rules and strategies for those graphs. We also define Database Port Graphs, to represent a relational database schema $\mathcal{DB} = (\mathcal{R}(\mathcal{A}), \mathcal{FD}, \mathcal{CK})$ (see section 2.1).

3.1 A Visual Domain Specific Language for Database Modelling

The visual building blocks of the language correspond to those of relational databases. Port graph nodes will have an attribute DbType whose value indicates the role of the node. To avoid confusion, we will use Proper Case for relational database concepts (e.g., Attribute) and lower case for port graph concepts (e.g., attributed port graph).

We note here that an Attribute can occur in multiple relations. To this end, we can define a conceptual attribute node. Then we can define an attribute occurrence node to distinguish between appearances in different Relations. In this work, from now on, we only use occurrences and we call them Attribute.

Definition 3 (Relational Database Port Graph VDSL, RDPG-VDSL). *A* Relational Database Port Graph VDSL *is an attributed port graph* $G_{RDB} = (V, P, E, D)_{\mathcal{F}}$*, such that* V *includes the following disjoint sets of nodes (as well as application specific nodes):*

- V_R*: relation nodes (*DbType *= REL);*
- V_A*: attribute nodes (*DbType *= ATTR);*
- V_{FD}*: functional dependency nodes (*DbType *= FD);*
- V_{CK}*: candidate key nodes (*DbType *= CK).*

P includes the following disjoint sets of ports:

- P_{ATT}*: contained attribute ports pATT;*
- P_{REL}*: parent relation ports pREL;*
- P_{DA}*: dependency attribute ports pFD;*
- P_{FD}*: functional dependency ports pFDLHS and pFDRHS;*
- P_{CK}*: relation candidate key ports pCK;*
- P_{KEY}*: (candidate) key attribute ports pKEY.*

and the functions Attach and Connect are such that:

- *if* $p \in P_{ATT}$*,* $Attach(p) \in \{V_R \cup V_{CK}\}$*;*
- *if* $p \in P_{REL}$*,* $Attach(p) \in \{V_A \cup V_{CK}\}$*;*
- *if* $p \in P_{DA}$*,* $Attach(p) \in V_A$*;*
- *if* $p \in P_{FD}$*,* $Attach(p) \in V_{FD}$*; each node in* V_{FD} *has two ports, pFDLHS and pFDRHS;*
- *if* $p \in P_{CK}$*,* $Attach(p) \in V_R$*;*
- *if* $p \in P_{KEY}$*,* $Attach(p) \in V_A$*.*

Connect includes the following pairs of ports (and associated edges):

- Functional Dependency: (pFD, pFDLHS) and (pFDRHS, pFD), *where pFD* $\in P_{DA}$ *and pFDLHS, pFDRHS* $\in P_{FD}$*. Given a dependency* $\varphi : X \to A$ *the pFD port of every attribute node corresponding to* X *will be connected to the pFDLHS port of the dependency node corresponding to* φ *and the pFDRHS port of the FD node representing* φ *will be connected to the pFD port of the attribute node representing* A*.*

- Attribute in relation: (pATT, pREL), *where* $pATT \in P_A$ *and* $pREL \in P_{REL}$. *Given a relation* R_i *and its attribute* A*, the* $pATT$ *port of the node representing* R_i *will be connected to the* $pREL$ *port of the node representing* A.
- Attribute in candidate key: (pATT, pKEY), *where* $pATT \in P_A$ *and* $pKEY \in P_{CK}$. *Given a candidate key* CK_i *and every attribute* $A_j \in CK_i$*, the* $pKEY$ *port of the node corresponding to* A_j *will be connected to the* $pATT$ *port of the node corresponding to* CK_i.
- Candidate Key of Relation: (pREL, pCK), *where* $pREL \in P_{REL}$ *and* $pCK \in P_{CK}$. *Given a relation* R_i *and its candidate key* CK_j*, the* pCK *port of the node representing* R_i *will be connected to the* $pREL$ *port of the node representing* CK_j.

As a particular case of the above defined class, we now define the Database Port Graph (DBPG) that represents $\mathcal{DB} = (\mathcal{R}(\mathcal{A}), \mathcal{FD}, \mathcal{CK})$. Most importantly, we constrain that one Relation is represented by only one relation node and similarly, one FD is represented by only one FD node. Also, each Attribute occurrence is represented by one attribute node. This design decision is based on the separation of concerns principle.

Definition 4 (Database Port Graph, DBPG). *A* Database Port Graph *is an RDPG such that the following constraints are satisfied:*

- V_R*: one node* DbType = REL *per Relation schema in* $\mathcal{R}$*;*
- V_A*: one node* DbType = ATTR *per Attribute occurrence in any of the* R_i*;*
- V_{FD}*: one node* DbType = FD *per Functional Dependency in* $\mathcal{FD}$*;*
- V_{CK}*: one node* DbType = CK *per Candidate Key in* $\mathcal{CK}$.

The Functional Dependency Port Graph (FDPGs) [27] is a particular case of the above defined Database Port Graph, with Attribute and FD nodes only. An example FDPG is given in Figure 10 in Appendix E.

3.2 Variadic Rewriting Rules

To deal with functional dependencies of various arities, previous works used multiple rules [27] or internal data structures (e.g. compound node in [5]). Here, we present an extension to the port graph rewriting rule language, called variadic rewriting rules (VRRs), inspired by Variadic Interaction Nets [18]. A variadic rule represents a family of rules that differ only in the number of times a subgraph is repeated. First, we propose a container structure that clearly identifies in a port graph the subgraph that will be repeated.

Definition 5 (Pattern Container). *A* pattern container *is a subgraph within a port graph such that if an edge links two ports that belong to the container, the edge also belongs to the container.*

A pattern container has two attributes: a name*, and a* multiplicity *that specifies the maximum number of times the encapsulated pattern will be repeated.*

Edges that connect a port in a pattern container and a port in the outside graph are called variadic edges*. Variadic edges also have an attribute* multiplicity *to control the number of repetitions.*

Definition 6 (Variadic Port Graph Rewrite Rule, VRR). *A* variadic port graph rewrite rule*, denoted $L \Rightarrow^{\mathcal{V}} R$, is a port graph rewrite rule with at least one* pattern container *on the LHS. Multiple pattern containers must not overlap. Pattern container names must be unique on the LHS.*

Given a VRR, we obtain its family of rules by running the Expansion algorithm defined below.

Definition 7 (Variadic Pattern Expansion). *For each pattern container, we generate i copies, where i is the value of the* multiplicity *attribute, as follows:*

1. ***Synchronized Expansion:***
   If a pattern container is present on both sides of a VRR (i.e. their names are identical), then the pattern is expanded in an iterative way on both sides of the rule until i number of copies of the encapsulated subgraph are generated. If variables are used in attributes then a different variable should be used in each copy. The expansion iterator works pairwise; that is, not all combinations of expansions are generated on LHS and RHS, but only the same number of repetitions on the two sides.
2. ***LHS-only Expansion:***
   If the pattern container is defined on LHS only, the expansion happens on LHS only, in the same iterative way.

If multiple patterns are defined, they are expanded independently, i.e. all combinations are generated, by nested iteration. Generally, the order in which the combinations are generated, does not matter.

A variadic edge is expanded based on the value j of its multiplicity attribute. If it equals the multiplicity i of the container it belongs to, then it is fully *expanded, i.e. it is created in all i instances of the container. A* partially *expanded variadic edge ($j < i$) is only created in the first $1 \ldots j$ instances. In other words, in the* n*th iteration of the expansion of the pattern container the variadic edge belongs to, if $n < j$ then n copies are created; otherwise j copies are created.*

In Porgy, which does not have a mechanism to define variadic rules, Definition 7 can be implemented as a macro expansion. The pattern container is visually represented by an enclosing rectangle (a metanode): attributes *name* and *i* are displayed at the top of the rectangle; and, on the LHS, a + sign in its upper-right corner as shown in the example below (Figures 1 and 2). Fully expanded variadic edges are also marked with a + sign over them and partially expanded variadic edges have the attribute value j displayed over them.

In this example, because the pattern appears in both sides, the expansion will generate a rule version with one node Y, another with 2 and another with 3, we show in the picture just the version corresponding to 3. If the node Y has an attribute a whose value is an expression containing variables, for example x, then each copy of the node Y will have attribute a with values x_1, x_2, x_3.

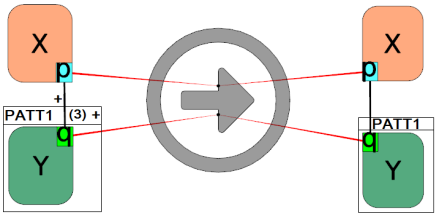

Fig. 1: VRR example

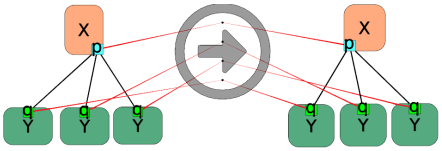

Fig. 2: VRR example expanded, $i = 3$

4 Computing the Syntactic Closure of Σ

Given an FDPG representing set of functional dependencies Σ, we compute its syntactic closure by applying the rules Reflexivity, Augmentation and Transitivity, defined below, controlled by Strategy 1: Syntactic Closure. From now on, we colour-code nodes, as a visual aid. Attribute nodes are green, FD nodes are purple and ports are dark blue. We use other colours for highlighting purposes.

4.1 Rewriting Rules

In the rules below, $x, y, \ldots$ represent name variables for attribute nodes, and $f1, f2, \ldots$ are name variables for FD nodes.

Augmentation. The Augmentation rule (see Figure 3) finds every attribute node y that doesn't have an edge into the FDLHS port of $f1$, regardless of what other attributes are connected there already. We find all non-connected attribute nodes by using the anti-edge feature [26] of PORGY. An anti-edge is represented by a grey double line in the rule editor. The matching algorithm deems the candidate sub-graph isomorphic if no edge is to be found between the two ports connected by the anti-edge.

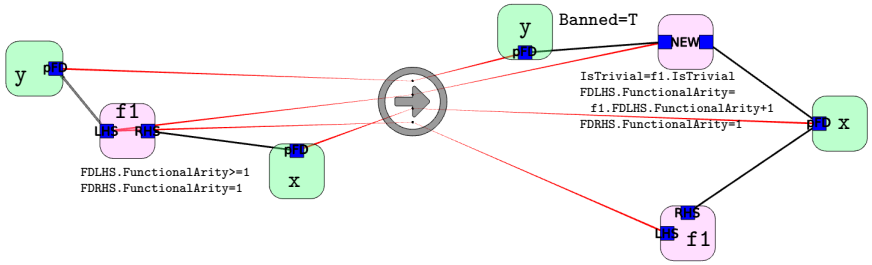

Fig. 3: Augmentation rule.

The rule creates a new FD node and assigns all pre-existing attributes and y to the left side of $f1$. The original $f1$ dependency node is also kept. We use bridge ports (red edges) to keep and copy the already existing edges into the FDLHS ports of $f1$ and *NEW*. If $f1$ was trivial then the new dependency will also be marked trivial. The rule also increases the *FunctionalArity* counter by 1 indicating that a new attribute is connected to the FDLHS port.

Reflexivity. The Reflexivity rule (Figure 4) applies to a node representing the attribute x and generates a trivial dependency $x \to x$. Then the attribute node x is banned so the rule cannot apply again on the same attribute.

The red edges in the arrow node indicate that when applying the rule, any edges connected to the **pFD** port of x in the left-hand side should be transferred to the corresponding **pFD** port of x in the right-hand side.

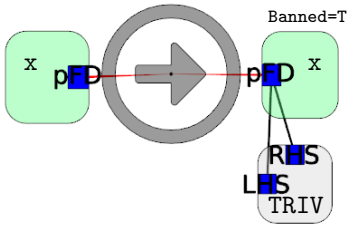

Fig. 4: Reflexivity rule.

Transitivity. A family of Transitivity rules was described in [27] to detect transitive functional dependency chains $f_1 : X \to Y$ and $f_2 : Y \to A$. Instead, here we provide a compact representation of the transitivity axiom in the form of a variadic rule, shown in Figure 5. This rule subsumes the family of Transitivity rules used in previous work.

As mentioned in Section 3.2, $|Y| = k \geq 1$ means that f_1 turns into a set of dependencies $f_1^1 \dots f_1^k$. The connections between the pFD ports of X attribute nodes and the pFDLHS ports of $f_1^1 \dots f_1^k$ nodes have to be preserved as well as copied onto the pFDLHS port of the newly created FD node (called NEW in Figure 5). We achieve this without needing to include the attribute nodes representing X in the rule, thanks to the bridge ports of the arrow node and the connecting red edges, as explained in Definition 6. Then, to cover all cases, we define a VRR pattern over f_1 and Y, with $i = 1 \dots k$. By definition the bridge ports, red edges and the normal edges into y_1.pFD, $\dots$, y_k.pFD will be repeated during the expansion.

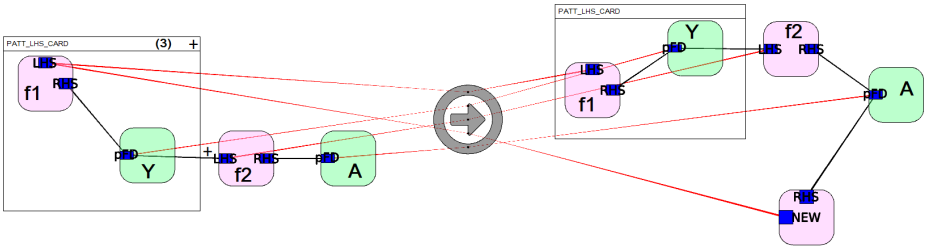

Fig. 5: Variadic Transitivity rule.

We show an example expansion of the Transitivity VRR to Transitivity-3, i.e. with 3 repetitions, on Figure 11 in Appendix E. FD nodes are labelled by records containing an attribute UID that uniquely identifies the Functional Dependency, except for trivial dependencies that are all given UID = 1. Nodes representing non-trivial FDs are given a prime number as UID. This offers extra, domain-specific backtracking functionality for dependencies, as explained below.

Note that the Reflexivity, Augmentation and Transitivity rules never remove the matching subgraph. Therefore, these rules could run for ever. To prevent this, we use conditional rules and focusing constructs in Section 4.2 to define the Syntactic Closure strategy. To ensure that the iteration of the Transitivity rule terminates when no new transitive dependencies can be inferred, we use

the UID attribute of FD nodes. When the Transitivity rule creates a transitive dependency node, it multiplies the UIDs of the contributing FDs and assigns the result as UID of the new FD node. We forbid the application of the rule if a node already exists with that UID (using `NotNode()` in the rule condition).

4.2 Syntactic Closure Strategy

The Strategy 1: Syntactic Closure applies first the Reflexivity rule as much as possible in the current graph. Each application bans an Attribute node, which ensures termination since matching is not allowed on banned nodes.

In lines 5-6, we set the Position subgraph to be the whole graph and the Banned subgraph to empty. Then, while there is at least one FD node the Augmentation rule hasn't *visited* and *iterated*, `one`($AugIterOn$) sets $AugIter$ and $AugVisit$ flags to true on a randomly selected FD node. The Augmentation rule will be applied on this FD node and all attribute nodes that are not connected to the FDLHS port of said FD node. Every attribute node used by this rule is banned to prevent re-application. Once all possible applications are processed, $AugIter$ flag is set to false and the Banned subgraph to empty. The iteration proceeds to the next, not yet visited, FD node. All new FD nodes, created by the Augmentation rule, are assigned a unique UID using the `update()` construct to call the function, `GenerateNextPrime()`, using PORGY's Python API.

Strategy 1: Syntactic Closure

```
1  //——— Reflexivity ———
2  setPos(all(crtGraph));
3  repeat(one(Reflexivity));
4  //——— Augmentation ———
5  setPos(all(crtGraph));
6  setBan(all([emptySet]));
7  while(match(AugIterOn))do(
8     one(AugIterOn);
9     repeat(one(Augmentation));
10    one(AugIterOff);
11    setBan(all([emptySet]))
12 );
13 update("GenerateNextPrime"{result : UID});
14 //——— Transitivity ———
15 while(match(IterOn))do(
16    one(IterOn);
17    repeat(one(Transitivity);#Augmentation#);
18    update("GenerateNextPrime"{result : UID});
19    one(IterOff)
20 );
21 repeat(one(ResetVisitedFlags));
22 //——— Cleanup ———
23 repeat(one(Cleanup))
```

Next, we compute transitive dependencies (lines 15-21), calling the Augmentation strategy (lines 5-12) after each application of the Transitivity variadic rewriting rule in line 17.

Since the rewrite rules may generate functional dependencies that already exist in the graph (despite the condition in the Transitivity rule), we add CleanUp rules to remove duplicates: see the Cleanup VRR in Figure 12 and an example expansion in Figure 13, both in Appendix E.

4.3 Example of application

Our strategy computed the syntactic closure of $\Sigma = \{AB \to C, ABC \to D\}$; the resulting FDPG can be seen on Figure 6.

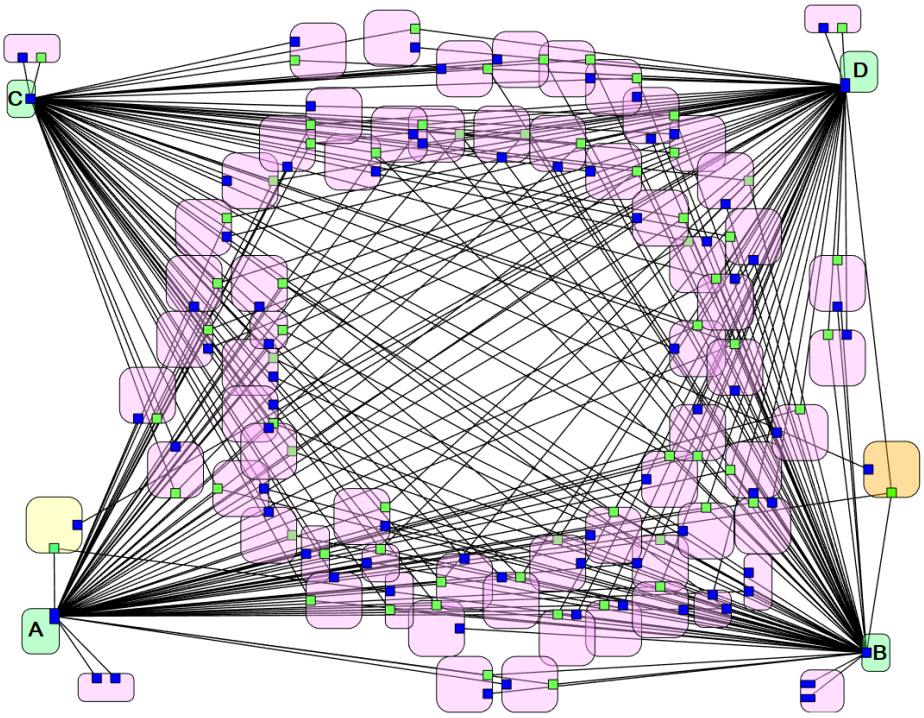

Fig. 6: Syntactic closure of $\Sigma = \{AB \to C, ABC \to D\}$.

Attributes A, B, C and D and their trivial dependencies can be seen in the four corners of the graph. As an example, we highlighted two FD nodes. The first one, in orange on the right hand side of the image, represents the dependency $ABD \to C$ which was created by augmenting $AB \to C$ with D. The second one represents $AB \to D$. This FD, shown on the left side of Figure 6 in yellow, was found by the Transitivity rule, matching on dependencies $A \to A, B \to B, AB \to C$ and $ABC \to D$.

4.4 Visual Analysis of the Closure

We now turn our attention to usual questions about Σ^+. For example, using the derivation tree in PORGY, it is possible to track how and when a particular dependency was generated: If we alter the colour of any FD node in a leaf node of the derivation tree, PORGY will back-propagate this change up the tree. This way, we can identify the exact step where the FD was created, and by zooming on the edges of the derivation tree we can see which of Armstrong's axioms generated the dependency.

Strategy 2: Membership Problem

```
1  setPos(all(
2  property(crtGraph, port,DbType == "FDLHS" && FunctionalArity == 3)
3  ∩ ngb(property(crtGraph, node,DbType == "ATTR"
4        && viewLabel == "A"), edge,DbType == "L")
5  ∩ ngb(property(crtGraph, node,DbType == "ATTR"
6        && viewLabel == "B"), edge,DbType == "L")
7  ∩ ngb(property(crtGraph, node,DbType == "ATTR"
8        && viewLabel == "D"), edge,DbType == "L")
9  ∩ ngb(property(crtGraph, node,DbType == "ATTR"
10       && viewLabel == "C"), edge, DbType == "R") ) ); //end setPos
11 (isEmpty(crtPos))orelse(repeat(one(Highlight)))
```

Another important question in database design is the *Membership Problem* [8]: given a set of FDs Σ, and an FD, φ, determine if $\varphi \in \Sigma^+$. Two groups of algorithms were developed to solve the membership problem: 1. generate a syntactic closure and check if $\varphi : X \to A$ is in it, or 2. compute the closure of X, X^+ and check if A is in it. Following the first approach, we can solve the problem by running a strategy to find and highlight the FD node that represents φ in the syntactic closure, if it exists, and fail if $\varphi \notin \Sigma^+$. For example, Strategy 2 was used to find the dependency $ABD \to C$, highlighted in Figure 6 (due to space constraints, we refer the reader to [14] for explanations of the constructs used). Following the second approach, we can simply use Strategy 1 but focusing on the set X of attributes. We only need to replace the expression `setPos(all(crtGraph))` with one that describes X, then the rewriting steps will apply on the attribute set in question. For example, if $X = B$, we write `setPos(all(property(crtGraph, node`, DbType ==”ATTR” && viewLabel==”B”))).

4.5 Correctness

The strategic program $Cl = [\mathcal{F}, closure]$ consisting of an initial port graph $\mathcal{F}$ representing a set of functional dependencies Σ, and the syntactic closure strategy defined in Strategy 1, correctly computes the syntactic closure Σ^+. Proofs of the propositions stated below are given in the Appendix.

Proposition 1 (Termination). *For any initial FDPG $\mathcal{F}$, the strategic program $Cl = [\mathcal{F}, closure]$ terminates.*

To prove the correctness of our program, we first show that the three rules Reflexivity, Augmentation and Transitivity are sound and complete, that is, given Σ, we can compute Σ^+ by using these three rules.

Proposition 2 (Soundness and Completeness of the Rules). *The Reflexivity, Augmentation and Transitivity rules stated below are sound and complete:*

1. *Reflexivity: for any attribute A, $A \to A$.*
2. *Augmentation: If $X \to A$ then $XY \to A$ for any attribute A and sets X, Y of attributes.*
3. *Transitivity: If $X \to A_i$ $(1 \leq i \leq n)$ and $A_1, \ldots, A_n \to B$ then $X \to B$.*

Since the Reflexivity, Augmentation and Transitivity port graph rewriting rules implement the rules stated in Proposition 2, to prove that Cl is sound and complete it suffices to show that any sequence of applications of these three rules can be transformed into a sequence in the order defined by the closure strategy.

Definition 8 (Canonical Form). *A sequence of applications of Reflexivity, Augmentation and Transitivity is in canonical form if it consists of applications of Reflexivity, followed by Augmentation, followed by Transitivity and Augmentation:* $(Reflexivity)^*(Augmentation)^*(Transitivity; Augmentation^*)^*$

Proposition 3 (Soundness and Completeness of the Strategy). *Canonical sequences are sound and complete.*

5 Finding a Minimal Cover

To generate a Minimal Cover (see Section 2 for the definition) we have to ensure that there are no extraneous attributes on FD left sides and there are no redundant FDs (we already have singleton right sides). Standard algorithms check this by running the Membership algorithm on an altered Σ: remove $X \to A$ and replace it with $X \setminus B \to A$. If this still yields the same Σ^+ then B is extraneous in X. Instead, since we have Σ^+ at hand, it suffices to check if there exists $Z \subset X \to A$, for all proper subsets Z. We use a variadic rewriting rule (Section 3.2) to specify a family of rules for every possible subset pair (n, k), where $n = |X|$ and $k = |Z|$. Since $Z \subset X$, we make use of the partially expanded variadic edge feature by restricting one variadic edge to only k expansions. We show the variadic rule (parameterised already with $n = 3, k = 2$) in Figure 7 (and an expansion in Figure 14 in Appendix E).

Next, we have to remove redundant FDs. A functional dependency $\varphi : X \to A$ is redundant, if $(\Sigma \setminus \varphi)^+ = \Sigma^+$ [21]. Previously published algorithms detected this by running a Membership check on $(\Sigma \setminus \varphi)$ to see if it yields φ. We note that in an FDPG representing $(\Sigma \setminus \varphi)^+$ there is an FDPG-Path from X to A. This path exists as an FD node created by the Syntactic Closure strategy, and

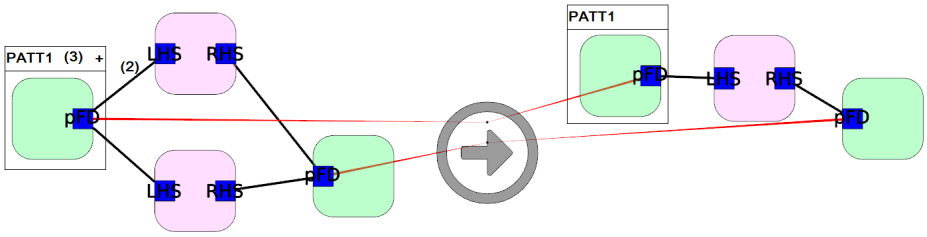

Fig. 7: Extraneous variadic rule.

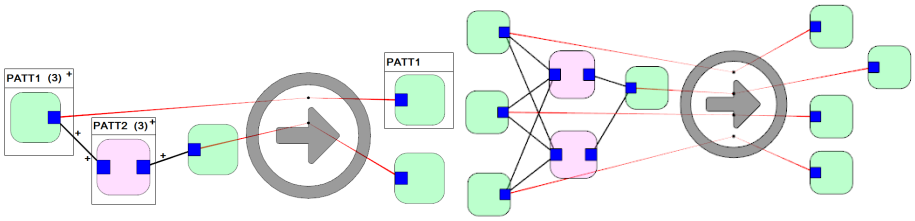

Fig. 8: Nonredundancy VRR.

Fig. 9: Nonredundancy rule, $i_1 = 3, i_2 = 2$.

if a dependency can be inferred in multiple ways, it is present multiple times. Since $|X| \geq 1$, we use a VRR to detect and remove the redundant FD nodes. We present the rule and an expansion in Figures 8 and 9.

Using the Extraneous and Nonredundancy rules, Strategy 3 computes a Minimal Cover. We reuse the Syntactic Closure strategy, but without the Clean Up rules. The Nonredundancy rule gets rid of duplicates. Lastly, we remove trivial dependencies (FD nodes where UID = 1), as they are not in the minimal cover.

Strategy 3: Minimal Cover

```
1 #Syntactic Closure without Cleanup#
2 setPos(all(crtGraph)); setBan(all([emptySet]));
3 repeat(one(Extraneous)); repeat(one(Nonredundancy));
4 repeat(one(RemoveTrivial));
```

6 Conclusion and Future Work

We introduced variadic rewriting rules and used these rules to define strategies that compute and analyse the syntactic closure of a set of Functional Dependencies. We have shown that these strategies are terminating, sound and complete. We have also defined additional rules and a strategy to compute Minimal Covers. A minimal cover is the input of algorithms to find candidate keys [24] and of Bernstein's 3NF Synthesis Algorithm [10]. The strategies that find these will make use of the already defined CK and Relation nodes. Furthermore, 3NF Relations will require the introduction of the notion of Foreign Key.

References

1. Abrial, J.: Data semantics. In: Klimbie, J.W., Koffeman, K.L. (eds.) Data Base Management, Proceeding of the IFIP Working Conference Data Base Management, Cargèse, Corsica, France, April 1-5, 1974. pp. 1–60. North-Holland (1974)
2. Andrei, O.: Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems. Ph.D. thesis, Institut National Polytechnique de Lorraine, Nancy, France (November 2008)
3. Andrei, O., Fernández, M., Kirchner, H., Melançon, G., Namet, O., Pinaud, B.: Porgy: Strategy-driven interactive transformation of graphs. In: Echahed, R. (ed.) TERMGRAPH. EPTCS, vol. 48, pp. 54–68 (2011)
4. Armstrong, W.W.: Dependency structures of data base relationships. In: Information processing 74: proceedings of IFIP Congress 74. pp. 580–583. North-Holland, Amsterdam (1974)
5. Ausiello, G., D'Atri, A., Saccà, D.: Graph algorithms for functional dependency manipulation. J. ACM **30**(4), 752–766 (1983). https://doi.org/10.1145/2157.322404
6. Badia, A., Lemire, D.: A Call to Arms: Revisiting Database Design. SIGMOD Rec. **40**(3), 61–69 (Nov 2011). https://doi.org/10.1145/2070736.2070750
7. Batini, C., D'Atri, A.: Rewriting systems as a tool for relational data base design. In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph-Grammars and Their Application to Computer Science and Biology, International Workshop, Bad Honnef, October 30 - November 3, 1978. Lecture Notes in Computer Science, vol. 73, pp. 139–154. Springer (1978). https://doi.org/10.1007/BFb0025717
8. Beeri, C., Bernstein, P.A.: Computational problems related to the design of normal form relational schemas. ACM Trans. Database Syst. **4**(1), 30–59 (1979)
9. Beeri, C., Fagin, R., Howard, J.H.: A complete axiomatization for functional and multivalued dependencies in database relations. In: Smith, D.C.P. (ed.) SIGMOD Conference. pp. 47–61. ACM (1977)
10. Bernstein, P.A.: Synthesizing third normal form relations from functional dependencies. ACM Trans. Database Syst. **1**(4), 277–298 (1976)
11. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P., Ringeissen, C.: An overview of ELAN. Electr. Notes Theor. Comput. Sci. **15**, 55–70 (1998). https://doi.org/10.1016/S1571-0661(05)82552-6, `https://doi.org/10.1016/S1571-0661(05)82552-6`
12. Ehrig, H., Engels, G., Kreowski, H., Rozenberg, G. (eds.): Handbook of graph grammars and computing by graph transformation: Applications, Languages and Tools, vol. 2. World Scientific (1999)
13. Embley, D.W., Mok, W.Y.: Mapping Conceptual Models to Database Schemas. In: Embley, D.W., Thalheim, B. (eds.) Handbook of Conceptual Modeling, vol. XIX, pp. 123–164. Springer (2011)
14. Fernández, M., Kirchner, H., Pinaud, B.: Strategic Port Graph Rewriting: an Interactive Modelling Framework. Mathematical Structures in Computer Science pp. 1–48 (Aug 2018). https://doi.org/10.1017/S0960129518000270, `https://hal.inria.fr/hal-01251871`
15. Fernández, M., Kirchner, H., Pinaud, B., Vallet, J.: Labelled graph strategic rewriting for social networks. J. Log. Algebr. Meth. Program. **96**, 12–40 (2018). https://doi.org/10.1016/j.jlamp.2017.12.005, `https://doi.org/10.1016/j.jlamp.2017.12.005`

16. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems - the complete book (2. ed.). Pearson Education (2014)
17. Jahnke, J.H., Zündorf, A.: Applying Graph Transformations to Database re-engineering. In: Ehrig et al. [12], pp. 267–286
18. Jiresch, E.: Extending the interaction nets calculus by generic rules. In: Alves, S., Mackie, I. (eds.) Proceedings 2nd International Workshop on Linearity, LINEARITY 2012, Tallinn, Estonia, 1 April 2012. EPTCS, vol. 101, pp. 12–24 (2012). https://doi.org/10.4204/EPTCS.101.2, https://doi.org/10.4204/EPTCS.101.2
19. Kalleberg, K.T.: Stratego: a programming language for program manipulation. ACM Crossroads **12**(3), 4 (2006). https://doi.org/10.1145/1144366.1144370, https://doi.org/10.1145/1144366.1144370
20. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theor. Comput. Sci. **109**(1&2), 181–224 (1993). https://doi.org/10.1016/0304-3975(93)90068-5, https://doi.org/10.1016/0304-3975(93)90068-5
21. Maier, D.: Minimum Covers in Relational Database Model. J. ACM **27**(4), 664–674 (1980). https://doi.org/10.1145/322217.322223
22. Maier, D.: The Theory of Relational Databases. Computer Science Press (1983)
23. Plump, D.: The design of GP 2. In: Escobar, S. (ed.) Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011. EPTCS, vol. 82, pp. 1–16 (2011). https://doi.org/10.4204/EPTCS.82.1, http://dx.doi.org/10.4204/EPTCS.82.1
24. Saiedian, H., Spencer, T.: An efficient algorithm to compute the candidate keys of a relational database schema. Comput. J. **39**(2), 124–132 (1996). https://doi.org/10.1093/comjnl/39.2.124
25. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. In: Ehrig et al. [12], pp. 551–603
26. Vallet, J.: Where Social Networks, Graph Rewriting and Visualisation Meet: Application to Network Generation and Information Diffusion. Ph.D. thesis, University of Bordeaux, France (2017), https://tel.archives-ouvertes.fr/tel-01691037
27. Varga, J.: Finding the Transitive Closure of Functional Dependencies using Strategic Port Graph Rewriting. In: Fernández, M., Mackie, I. (eds.) Proceedings Tenth International Workshop on Computing with Terms and Graphs, Oxford, UK, 7th July 2018. Electronic Proceedings in Theoretical Computer Science, vol. 288, pp. 50–62. Open Publishing Association (2019). https://doi.org/10.4204/EPTCS.288.5

Appendix A Proof of Proposition 1

Proposition (Termination)**.** *For any initial FDPG $\mathcal{F}$, the strategic program $Cl = [\mathcal{F}, closure]$ terminates.*

Proof. Since the strategy applies three strategies sequentially (Reflexivity, Augmentation and Transitivity), it is sufficient to show that each of them terminates. For this, we show that for each of them there is a measure which is strictly decreasing with respect to a well-founded ordering.

Reflexivity Strategy: The measure in this case is the number of non-banned nodes in the graph. Each application of the reflexivity rule (Figure 4 strictly decreases the number of non-banned nodes. Since the rule can only apply to non-banned nodes, the iteration defined in line 3 of the closure strategy terminates.

Augmentation Strategy: Two looping constructs are used in this strategy, a while-loop starting in line 7 and a repeat-loop in line 9. The while-loop is controlled by a condition `match`($AugIterOn$), which requires a non-visited node to succeed. The rule AugIterOn applied in line 8 sets the AugIter and AugVisit flags to true on a randomly selected FD node that has AugVisit flag = False. Each application of the Augmentation rule in the loop in line 9 bans the attribute node used, so the the repeat loop terminates, and afterwards the AugIter flag is set to false but the AugVisit flag remains untouched. This means that each iteration of the while loop in line 7 decreases the number of non-visited nodes, thus the strategy terminates.

Transitivity Strategy: This strategy, without the calls to the Augmentation strategy in line 17 and line 23, was shown to be terminating in [27]. We use the same measures, since they are not affected by the call to the Augmentation strategy: For `repeat`(`one`($Transitivity_k$); $\#Augmentation\#$), the measure is the number of possible matches of the LHS subgraph of f1. This is a good measure because a) the Rule Condition on UID in $Transitivity_k$ prevents re-application of the rule to the same nodes and b) even though the NEW node is added in the right-hand side, it is not part of the LHS subgraph of f1. Similarly, the nodes added by Augmentation are not connected to f1. By the time we call the Augmentation strategy from Transitivity, f1 will have been augmented and its AugVisit flag is permanently set to True. The only FD node Augmentation will be able to touch at this point is NEW.

For `while`(`match`($IterOn$))`do`(...) the measure is $|V_{FD}|_{G0} + |\Sigma^+| - |V_{FD}|_{Gi}$. That is, the number of FD nodes in the initial graph plus the size of the closure less the number of FD nodes after the i-th application of the loop. With one successful application of the Transitivity rule, the number of FD nodes increases therefore the measure decreases.

Termination of the CleanUp rule iteration is straightforward, since these rules decrease the size of the graph. □

Appendix B Proof of Proposition 2

Proposition (Soundness and Completeness of the Rules)**.** *The Reflexivity, Augmentation and Transitivity rules stated below are a sound and complete:*

1. *Reflexivity: for any attribute A, $A \to A$.*
2. *Augmentation: If $X \to A$ then $XY \to A$ for any attribute A and sets X, Y of attributes.*
3. *Transitivity: If $X \to A_i$ $(1 \leq i \leq n)$ and $A_1, \ldots, A_n \to B$ then $X \to B$.*

Proof. First, we observe that the rules are sound: they are particular cases of the axioms $A1$-$A3$ given in Section 2, which are sound and complete. Hence, it is sufficient to prove that the rules permit us to derive the axioms $A1$-$A3$.

Remark. Our rules assume that right sides of functional dependencies consist of only one attribute. Since the Union and Decomposition Axioms can be derived from $A1$-$A3$ and by them one can infer $X \to YZ$ from $X \to Y$ and $X \to Z$, and reciprocally, from $X \to YZ$ one can infer $X \to Y$ and $X \to Z$, it is sufficient to consider single attributes in right-hand sides of dependencies.

Axiom $A1$ can be derived as follows: Let $Y = \{A_1 \ldots A_n\}$ and assume $Y \subseteq X$. Using the Reflexivity rule, we can derive $A_i \to A_i$ and using Augmentation, we derive $A_i X \to A_i$ for each $A_i \in Y$. This is sufficient as explained in the remark above.

Axiom $A2$ can be derived as follows: Let
$Y = \{A_1 \ldots A_n\}$,
$Z = \{A_n, A_{n+1}, \ldots A_p\}$ and
$W = \{A_n, A_{n+1}, \ldots A_p, A_{p+1}, \ldots, A_k\}$
and assume $X \to Y$, that is, $X \to A_i$ for each $A_i \in Y$ as explained in the remark above. By repeated applications of the Augmentation rule, we derive $XW \to A_i$ for each $A_i \in Y$. Note that by Reflexivity, we can derive $A_j \to A_j$ for any $A_j \in Z$ and by repeated applications of Augmentation, we obtain $XW \to A_j$, since $A_j \in W$. Therefore $XW \to A_i$ for each $A_i \in YZ$, as required.

Axiom $A3$ is derived using the Transitivity rule: Assume $X \to Y$, that is, $X \to A_i$ for each $A_i \in Y$, and $Y \to B_i$ for each $B_i \in Z$. Using the Transitivity rule, we derive $X \to B_i$, as required. □

Appendix C Proof of Proposition 3

Proposition (Soundness and Completeness of the Strategy)**.** *Canonical sequences are sound and complete.*

Proof. By Proposition 2, Reflexivity, Augmentation and Transitivity are sound and complete. Assume a set Σ of functional dependencies is inferred using a non-canonical sequence S of applications of Reflexivity, Augmentation and Transitivity rules. We show that the steps of application of the rules can be reorder to obtain a canonical sequence that derives the same set of dependencies. Since applications of Reflexivity are independent of other rules, we can reorder the steps, to move all applications of Reflexivity to the start of the sequence, obtaining a sequence $S_1 = S_{Refl}S'$ that derives Σ and such that S_{Refl} contains only applications of the Reflexivity rule, and S' has no applications of Reflexivity.

Similarly, any application of Augmentation in S' on a dependency that exists in Σ or has been obtained by Reflexivity or by a previous Augmentation step can be moved towards the start of S', obtaining a sequence $S_{Aug}S''$ where S_{Aug} consists only of Augmentation steps and all the applications of Augmentation in S'' use a dependency obtained by transitivity.

The sequence $S_{Refl}SAugS''$ is therefore canonical and derives the same set of functional dependencies as S. It follows that canonical sequences are sound and complete. □

Appendix D DBPG and FDPG Examples

The functional dependency graph $\Sigma = \{AB \to C, ABC \to D\}$:

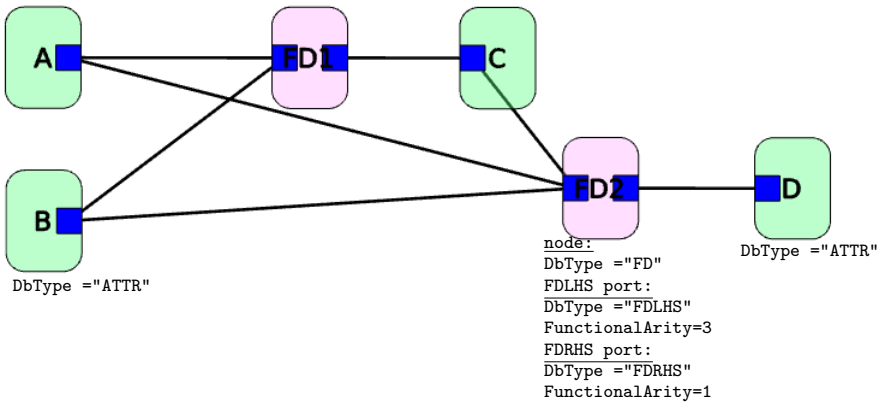

Fig. 10: The functional dependency graph $\Sigma = \{AB \to C, ABC \to D\}$.

Appendix E Variadic Rewriting Rule Expansion Examples

Example expansion of Transitivity VRR to Transitivity-3, where $i = 3$.

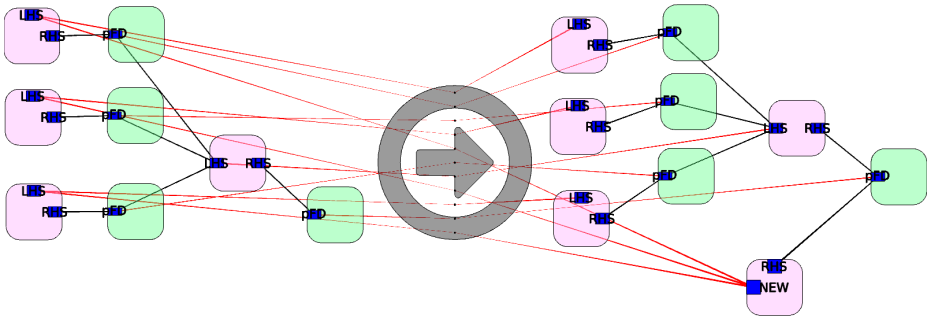

Fig. 11: An example expansion: Transitivity-3 rule.

The Cleanup variadic rewriting rule:

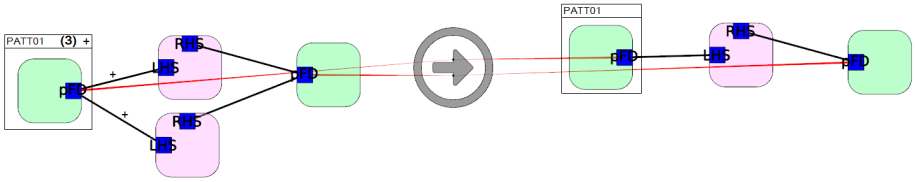

Fig. 12: Cleanup VRR.

Example expansion of Cleanup VRR to $i = 3$:

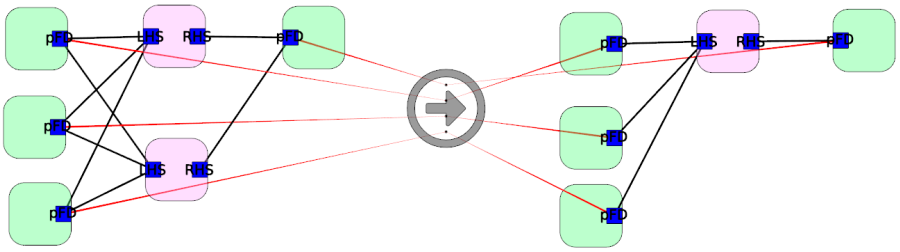

Fig. 13: An expansion of Cleanup VRR: Cleanup-3.

Example expansion of Extraneous VRR to $i = 2, j = 1$:

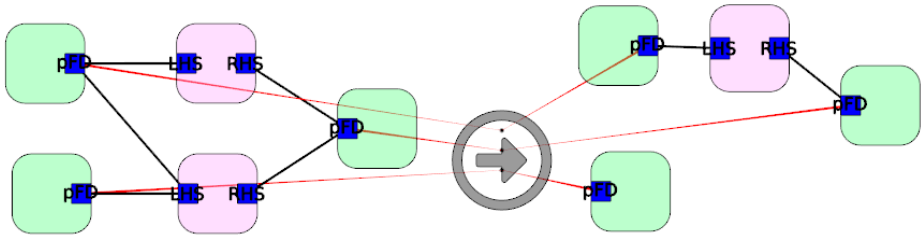

Fig. 14: Expanded extraneous rule, $i = 2, j = 1$.