Finding the Transitive Closure of Functional Dependencies using Strategic Port Graph Rewriting

János Varga
Department of Informatics
King's College London
janos.varga@kcl.ac.uk

We present a new approach to the logical design of relational databases, based on strategic port graph rewriting. We show how to model relational schemata as attributed port graphs and provide port graph rewriting rules to perform computations on functional dependencies. Using these rules we present a strategic graph program to find the transitive closure of a set of functional dependencies. This program is sound, complete and terminating, given the restriction that there are no cyclical dependencies in the schema.

1 Introduction

Traditionally, steps of relational database design include Conceptual, Logical and Physical modelling. The theory behind these steps (especially the first two) is well-understood and is part of the syllabus of many databases courses. Yet, database professionals often consider normalisation too cumbersome and do not apply normalisation theory, despite the clear advantages of normalised database designs. Badia and Lemire [4] also highlight that conceptual and logical models do not always carry enough information about database semantics thereby leading the architect to a sub-optimal design.

We use *attributed port graphs* to represent a relational schema and its semantics. Port graphs are graphs where edges are connected to nodes at specific points, called ports. They were introduced in [2] as a tool to model biochemical systems. In port graphs, nodes, edges and ports can have attributes, which are used to represent properties of the system modelled. In this paper we focus on using port graphs in the logical phase of database design. We show that port graphs are a good choice of data structure to store relational metadata and can be transformed without the loss of metadata.

To specify the transformations applied to relational schemata, we use *port graph rewriting systems*, a general class of graph rewriting systems [10]. The implementation framework we use is PORGY [12] – a visual, interactive tool for the specification, simulation and analysis of systems based on port graph rewriting. PORGY provides a graphical interface, where users can define a system and specify its dynamics by means of port graph rewrite rules and strategies. Port graphs have node, port and edge *attributes*, whose values are taken into account in port graph morphisms (used to define rewriting steps) and in strategy expressions (to control the application of rules). *Strategic graph programs* [12], consisting of an initial port graph and a set of rewrite rules controlled by a strategy, are the essence of PORGY. The strategy language offers separate primitives to select subgraphs of the model as focusing positions for rewriting and to select the rewrite rules to be applied, following the separation of concerns principle which makes programs easier to maintain and adapt. The strategy language also allows users to define strategies using not only operators to combine graph rewriting rules but also operators to deal with graph traversal and management of rewriting positions in a graph. PORGY provides a visual representation of

Submitted to:
TERMGRAPH 2018

the set of rewrite derivations (a *derivation tree*) and includes features such as cycle detection, to facilitate debugging.

We extend the rule language by adding the possibility to specify application conditions for a rule. That is, as part of a rewrite step, in the rule editor, users can define a set of conditions which are evaluated after a morphism has been found. The rewrite step is applied only if the rule condition evaluates to true. As a use-case we provide a set of port graph rewriting rules and a strategy to calculate the transitive closure of a set of functional dependencies. Although there are a number of tools already available to do the same, a distinctive advantage of our implementation is that it is visual and backtrackable (thanks to the derivation tree feature of PORGY). Our strategy is sound, complete and terminating, given the restriction that there are no cyclical dependencies in the schema.

Summarising, our contributions are:

1. a new visual language, based on port graphs, for logical design of relational schemata,
2. generic application conditions for rules (a port graph rewriting language extension),
3. a strategic graph program to find the transitive closure of a set of functional dependencies.

This last contribution is a key step towards building strategic graph programs to find minimal covers, candidate keys and Third Normal Form (3NF) relation schemata.

Related Work. Graph theory and graph rewriting is by no means a new addition to the set of tools that have been used for relational database design. In [11] hypergraphs are used and their well-formed property (called a canonical hypergraph) determines the quality of design they represent. The authors of [16] used directed graphs to find all candidate keys of a relation in polynomial time. A special family of labelled graphs, FD graphs, were introduced in [3] to obtain Σ^+ and Σ_{min}. In terms of graph transformations and rewriting we highlight two works. Hypergraph rewriting was used in [5] for the manipulation of functional dependencies and Triple Graph Grammars were used in [13] to optimize an already existing schema.

Organization. This paper is organised as follows. We briefly review relational database theory and port graph rewriting background in Section 2. We present our port graph visual language for logical design of relation schemata in Section 3. In Section 4 we define the syntax of the language for generic rule application conditions. Section 5 illustrates how the visual language and the generic rule application conditions can be used to find the transitive closure of a set of FDs. We then conclude in Section 6 by highlighting how these results can be used in future work. Due to space constraints proofs are omitted.

2 Background

In this section we will review the definitions and background in relational databases and port graph rewriting that we are going to use throughout this paper. Due to space constraints, for formal definitions and proofs, this section will refer the reader to the relevant works rather than recalling them. We will also briefly review related work that used graphs to represent or transform relational schemata.

2.1 Relational Database Design

We assume that the reader is familiar with the theory of logical design of relational databases [8, 9]. In particular, the definitions of: *relation schema*, *attribute*, *candidate key* and *functional dependency* (FD) [8, 9]. 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 X, Y, Z. This paper will denote the set of all FDs

of a relation schema Σ_R or just Σ, where appropriate. We also assume familiarity with the inference rules of functional dependencies, also known as Armstrong's Axioms [6]: Transitivity, Trivial Dependency, Augmentation, Union, Decomposition, Pseudotransitivity. We call the set of all FDs that can be inferred from Σ, using Armstrong's Axioms, the *syntactic closure* Σ^+. It was shown that Armstrong's Axioms are sound and complete, which means that they find only and all (respectively) semantically correct dependencies.

2.2 Port Graph Rewriting

An attributed port graph is a labelled attributed graph where nodes have specific points-of-connection called ports, and edges that are attached to ports. In this subsection we recall the most important port graph rewriting constructs from Sections 2 and 3 of [12], where the full formal definitions can also be found.

A *port graph rewrite rule* is a port graph $L \Rightarrow_{Where:=C} R$ consisting of two subgraphs L and R 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), a condition *Where* restricting the rule's matching, and ports to control the rewiring operations when rewriting steps are computed. Edges that run between ports of L, R and the arrow node are coloured red by PORGY to distinguish them from normal edges. We recall the definition of the *matching morphism* that states that 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 $saturated(p_1) \wedge ... \wedge saturated(p_n) \wedge B$. The predicate $saturated(g(p_i))$ is true if there are no edges between $g(p_i)$ and ports outside $g(L)$ in G. B is a Boolean expression such that all its variables occur in L.

Our contribution to the rewrite rule language is to provide the functionality of generic application conditions. This task was two-fold: we created a grammar for B and implemented a PORGY Rule Editor plug-in (called Rule Conditions).

We also recall here that a *strategic graph program* consists of a *located graph* (a port graph with two distinguished subgraphs that specify the locations where rewriting should take place or not), a set of *located rewrite rules*, and a *strategy expression*. In a located graph G^Q_P, P represents the *position* subgraph of G where rewriting steps may take place and Q represents the *banned* subgraph of G where rewriting steps are forbidden. A located rewrite rule $L \Rightarrow_C R^N_M$ can update P and Q in a rewrite step such that $P' = (P \setminus g(L)) \cup g(M)$ and $Q' = (Q \setminus g(L)) \cup g(N)$.

Our work to find the transitive closure is implemented in the form of a strategic graph program.

3 A Visual Language for Relational Schema Design

We now show how relational schemata (using functional dependencies only) can be modelled as attributed port graphs. We use the $'.'$ (dot, member-of) operator to refer to a particular port of a node.

We define attributes and FDs as nodes. The fact that an attribute belongs to the right- or left-hand side of a FD is represented by edges. However, when adding FDs to the visual language, we face a challenge. Because of the semantics of a FD (i.e. LHS determines RHS), strategic graph programs executed on this visual language have to be able to distinguish between LHS and RHS attributes of FDs. A non-trivial FD must have at least one attribute on both sides where (RHS $\not\subseteq$ LHS). Also, as per the separation of concerns principle, a FD has to be aware of the list of attributes on its sides, not the other way around.

Since port graphs are not typed graphs, firstly, we introduce an attribute called RelDbType which

denotes the role of the node in the relational context. Every new node (and all of its ports) created in a model have to have a *RelDbType* value. Attribute nodes have *RelDbType* = ATTR, FD nodes have *RelDbType* = FD. The port of an attribute that handles the connection to either side of a FD has *RelDbType* = pFD. The LHS and RHS connection ports of a FD node have *RelDbType* = FDLHS and FDRHS, respectively. Both FDLHS and FDRHS ports have an integer attribute *FunctionalArity* defined that allows the system to store the number of attributes on each side. This is required because the matching algorithm does not enforce exact arity since a particular port can be connected to other ports outside the match found, however, when matching on FDs and their LHSs, every single LHS attribute has to be in the matching subgraph.

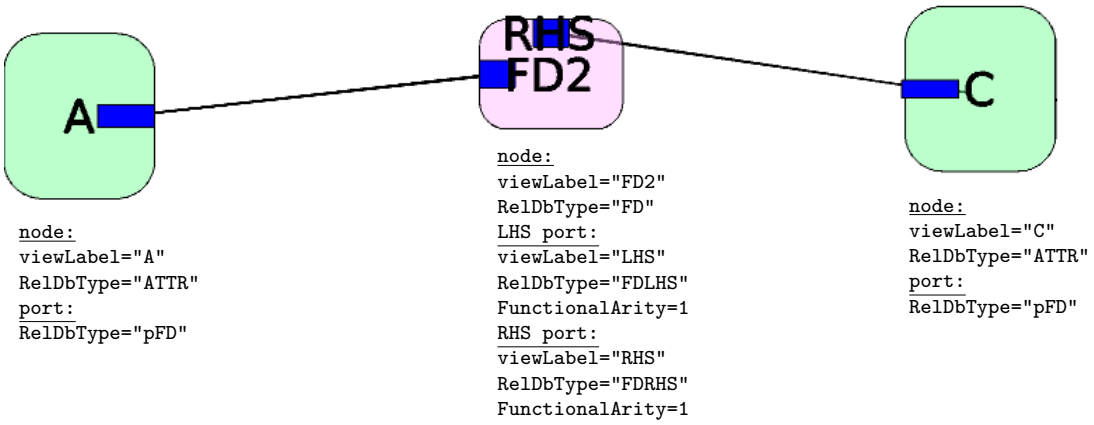

Figure 1: The functional dependency $A \rightarrow C$

Both FD and ATTR nodes have an integer UID attribute that allows the rules to assign a numeric identity value to them. This is useful when a rule adds a new FD and we want control over the value of the unique identifier. To assign meaningful names to nodes, we make use of the built-in *viewLabel* attribute. Figure 1 shows a functional dependency $A \rightarrow C$ as a port graph, with the relevant attribute values.

4 Generic Rule Application Conditions

We recall the structure of the arrow node *Where* attribute defined in Section 2.2. In this section we extend the rewrite rule language to provide the functionality of generic application conditions. Firstly, we define the context-free grammar for B, secondly, we present the PORGY Rule Editor plug-in (called Rule Conditions).

The EBNF grammar for B is defined in Figure 2. The structure of the grammar was inspired by C++ and follows the operator precedence of C++, too.

We point out that when referring to a `node` or `edge` the user has to use its internally assigned `id`. Also note that there is no `port` construct in the grammar – they have to be referred to as `node`. Terminal `number` can be any integer or floating-point number and `quoted_string` is an arbitrary-length string made up of letters, digits and symbols in double quotes. Similarly, `quoted_attribute_name` is a valid name of an attribute of `node`, `edge` or `port`.

We highlight the `NotNode()` operator: it iterates all nodes of G and checks if there exists a node with an attribute `quoted_attribute_name` and if the comparison on them evaluates to true. Intuitively, if

⟨*node*⟩ ::= '*n*(' `valid LHS node id` ')'

⟨*edge*⟩ ::= '*e*(' `valid LHS edge id` ')'

⟨*element attribute*⟩ ::= (⟨*node*⟩ | ⟨*edge*⟩) '.' `quoted_attribute_name`

⟨*factor*⟩ ::= `number` | `quoted_string` | '(' ⟨*expression*⟩ ')' | '!' ⟨*factor*⟩
| ⟨*element attribute*⟩ | 'max(' ⟨*expression*⟩ ',' ⟨*expression*⟩ ')'
| 'min(' ⟨*expression*⟩ ',' ⟨*expression*⟩')' | 'random('⟨*factor*⟩')'

⟨*term*⟩ ::= ⟨*factor*⟩ {('∗' | '/' | '%') ⟨*factor*⟩}

⟨*expression*⟩ ::= ⟨*term*⟩ {('+' | '−') ⟨*term*⟩}

⟨*comp operator*⟩ ::= '==' | '! =' | '>' | '<' | '> =' | '< ='

⟨*comparison*⟩ ::= ⟨*expression*⟩ ⟨*comp operator*⟩ ⟨*expression*⟩ |
'NotNode('`quoted_attribute_name` ⟨*comp operator*⟩ ⟨*expression*⟩')'

⟨*logical expression*⟩ ::= ⟨*logical term*⟩ { '||' ⟨*logical term*⟩ }

⟨*logical term*⟩ ::= ⟨*logical factor*⟩ { '&&' ⟨*logical factor*⟩ }

⟨*logical factor*⟩ ::= ⟨*comparison*⟩ | '!' ⟨*logical factor*⟩ | '(' ⟨*logical expression*⟩ ')'

⟨*rule condition*⟩ ::= {⟨*logical expression*⟩};

Figure 2: The rule application condition grammar.

at least one such node is found in G, `NotNode()` returns false. It is very important to note here that this check is performed on the entire graph G, not just in $g(L)$. This is fundamentally different from the rest of the rule application condition grammar, which only applies to $g(L)$. This is a consequence of the definition of the port graph rewrite rule which states that all variables in the Boolean expression of the *Where* attribute have to occur in L, so that the matching algorithm can work with them. When a match $g(L)$ is found, all variables of B are mapped so that their actual values can be found. However, when checking the absence of a node, we are not constrained by this, because we are not specifying a node in `NotNode()`– we are only specifying an attribute comparison that *must* evaluate to false on all the nodes of G.

PORGY offers a modular plug-in system allowing developers to create Python/C++ plug-ins. We implemented an LL-parser for the above detailed context-free grammar in C++ using the Boost Spirit Parser Framework. This framework generates and executes the parser design-time and builds and evaluates an abstract syntax tree run-time. When evaluated, the Boolean result is ANDed to the rest of the arrow node *Where* attribute by the matching algorithm. The parser ensures that all nodes, edges, ports and attributes referred to in the conditions exist on the LHS of the rule.

We also added a UI extension to PORGY that allows users to specify and parse/check the rule conditions. A screenshot of PORGY with the Conditions editor is presented in Figure 3. Examples of rule conditions can be found in Section 5.

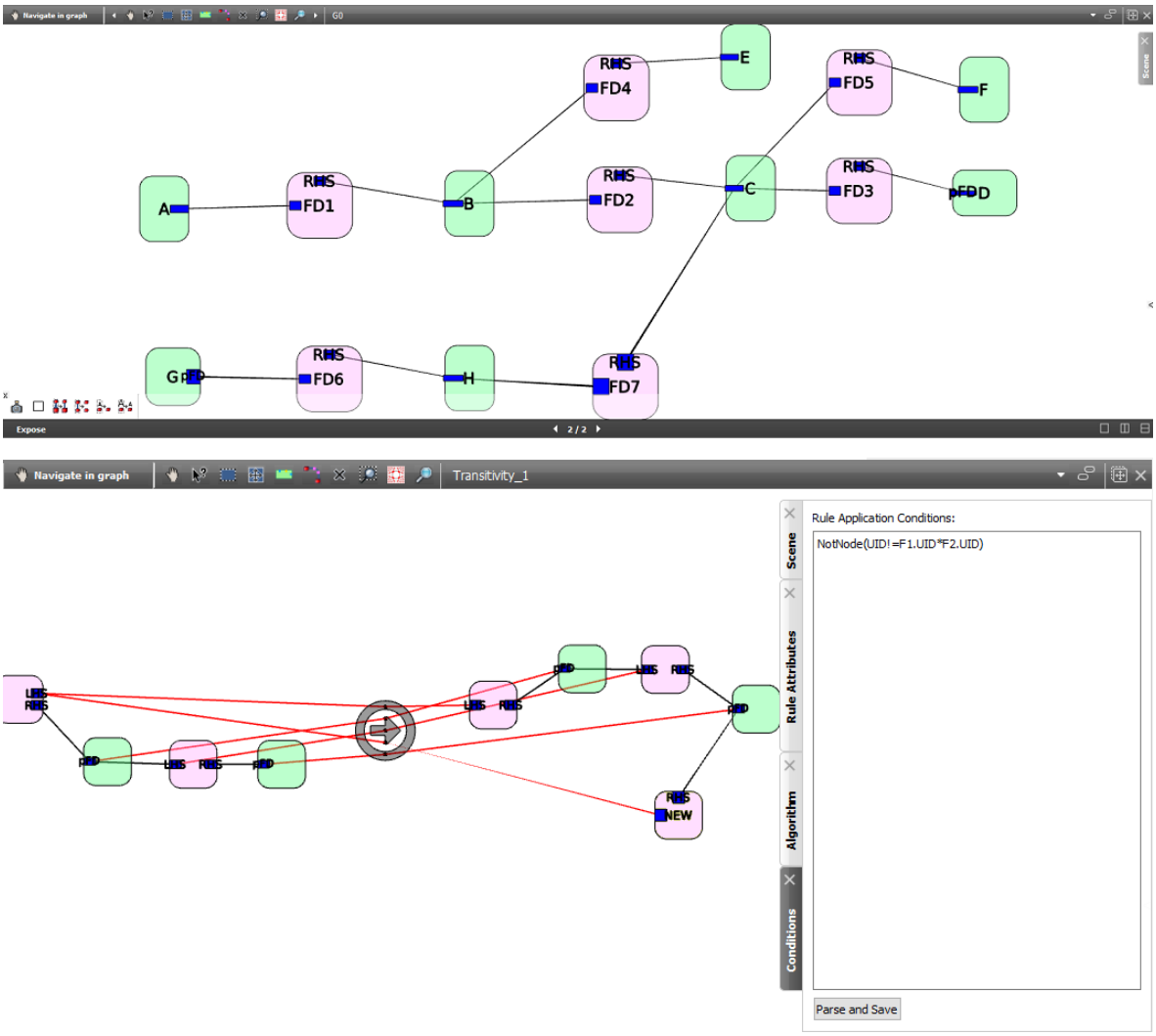

Figure 3: PORGY, an initial graph ($G0$), Transitivity$_1$ rule and the conditions editor.

5 Transitive Closure Strategy

In this section we present the strategy to find the transitive closure of a set of FDs using the previously introduced visual language and rule conditions. We work under the assumptions that a) FDs are in canonical form ie. every right-hand side is a single attribute and b) there are no cyclical dependencies.

The starting point of port graph rewriting, the original model, is referred to as G0. In our case, G0 has all user-defined FDs of a relation schema and follows the structure set out in Section 3. The task is to find a relational transitive closure of G0, i.e. to generate all the transitive dependencies.

Inspired by the Chase Algorithm [1, 15] we *iterate* every FD in G0 (and also those added by the rules), apply the rules that detect a transitive dependency pattern and create the new FD. Once we have every possible new transitive dependency that goes through the iterated FD, we mark it *visited*. We define two new Boolean node attributes : `iter` to flag the node currently being iterated and `visit` to permanently flag to node as visited. The two rules controlling the iteration are *IterOn* and *IterOff* (omitted).

IterOn randomly selects a node with attribute values `RelDbType=FD, iter=false, visit=false` and sets the two flags: `iter=true, visit=true`. *IterOff* rule selects the currently iterated FD node `RelDbType=FD, iter=true, visit=true` and turns the iteration flag off: `iter=false, visit=true`.

To track the maximum number of rule applications in which a particular ATTR can participate, we introduce the *MaxNoOfAppl* integer node attribute. In G0, *MaxNoOfAppl* = the number of FDs whose RHS is the ATTR node. Every time a Transitivity match is found such that ATTR is on the RHS of the new dependency, we increment the value by 1. Similarly, a new edge integer attribute is defined: *ApplCounter*. Its G0 value is 0 and it is incremented by 1 every time the edge is in $g(L)$ of any Transitivity rule. We add the *FDBanned* node attribute (Boolean), default value false, to be able to exclude a FD from the iteration or matching.

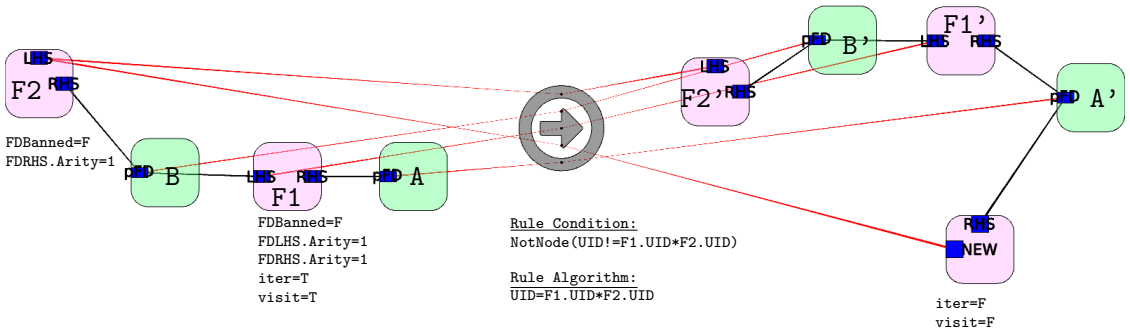

Figure 4: Transitivity$_1$ rule

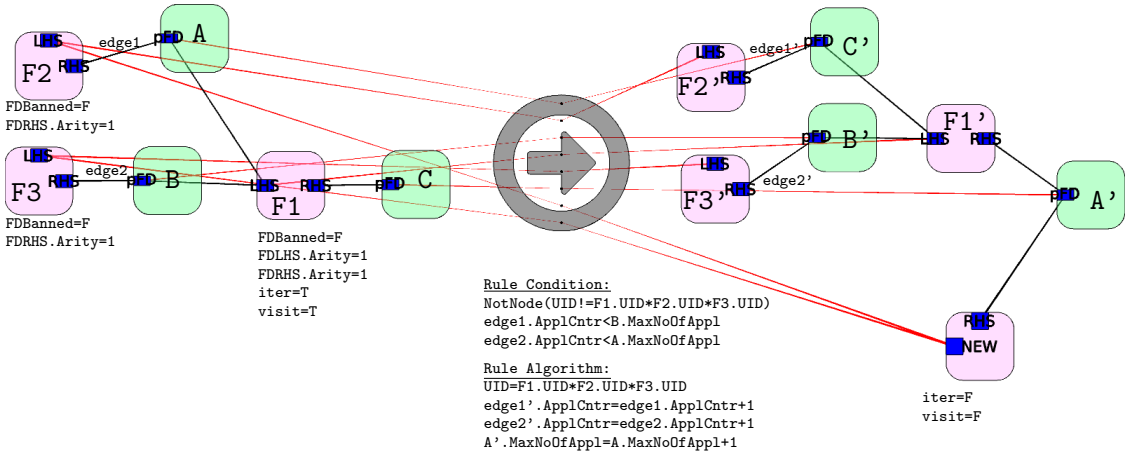

Figure 5: Transitivity$_2$ rule

The first two rules we created can be seen on Figures 4 and 5. Note that the node being iterated is the central one and its LHS subgraph increases in size as the *FunctionalArity* of it increases.

To offer extra, context-specific backtracking functionality, we assign prime numbers values to the UID attribute of every FD. When a new FD is created by any transitivity rule, the UID of the new FD is the product of the UIDs of the FDs that lead to the new FD. This is calculated and assigned by the Rule Algorithm feature of PORGY.

Finally, the strategy that uses the above defined rules is Strategy 1. The *ResetVisitedFlags* rule sets the flags `visit=0`, to allow for a whole new iteration of all FD nodes.

Strategy 1: Finding a Transitive Closure

```
while(match(IterOn))do(
  one(IterOn);
  repeat(one(Transitivity1));
  one(IterOff)
);
repeat(one(ResetVisitedFlags));
while(match(IterOn))do(
  one(IterOn);
  repeat(one(Transitivity2));
  one(IterOff)
)
```

We discuss the rewriting characteristics of Strategy 1 in Theorems 1, 2 and 3.

Theorem 1. *The Transitive Closure Strategy is sound. That is, it **only** finds functional dependencies that can be inferred from the original set using Armstrong's Axioms but ignoring the meaningless dependencies that would be generated by the Reflexivity and Augmentation axioms.*

Theorem 2. *The Transitive Closure Strategy is complete. That is, it finds **all** functional dependencies that can be inferred from the original set using Armstrong's Axioms but ignoring the meaningless dependencies that would be generated by the Reflexivity and Augmentation axioms.*

Theorem 3. *The Transitive Closure Strategy is a terminating program that never fails. That is, given the restriction that there are no cyclical dependencies in the original set, there is no infinite descending chain in the derivation tree.*

6 Conclusion

Our results can be used to build a strategic graph program that takes the transitive closure as input and finds a minimal cover [3, 14]. From a minimal cover, all candidate keys of a relation schema can be found [16]. The minimal cover and the set of candidate keys can then be used as inputs to Bernstein's Synthesis Algorithm to synthesize Third Normal Form [7] schemata.

References

[1] Alfred V. Aho, Catriel Beeri & Jeffrey D. Ullman (1979): *The Theory of Joins in Relational Databases*. *ACM Trans. Database Syst.* 4(3), pp. 297–314. Available at `http://doi.acm.org/10.1145/320083.320091`.

[2] Oana Andrei (2008): *Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems*. Ph.D. thesis, Institut National Polytechnique de Lorraine, Nancy, France.

[3] Giorgio Ausiello, Alessandro D'Atri & Domenico Saccà (1983): *Graph Algorithms for Functional Dependency Manipulation*. *J. ACM* 30(4), pp. 752–766, doi:10.1145/2157.322404. Available at `http://doi.acm.org/10.1145/2157.322404`.

[4] Antonio Badia & Daniel Lemire (2011): *A Call to Arms: Revisiting Database Design*. *SIGMOD Rec.* 40(3), pp. 61–69.

[5] Carlo Batini & Alessandro D'Atri (1978): *Rewriting Systems as a Tool for Relational Data Base Design*. In Volker Claus, Hartmut Ehrig & Grzegorz Rozenberg, editors: *Graph-Grammars and Their Application to Computer Science and Biology, International Workshop, Bad Honnef, October 30 - November 3, 1978*, *Lecture Notes in Computer Science* 73, Springer, pp. 139–154, doi:10.1007/BFb0025717. Available at `http://dx.doi.org/10.1007/BFb0025717`.

[6] Catriel Beeri, Ronald Fagin & John H. Howard (1977): *A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations*. In Diane C. P. Smith, editor: *SIGMOD Conference*, ACM, pp. 47–61. Available at `http://doi.acm.org/10.1145/509404.509414`.

[7] Philip A. Bernstein (1976): *Synthesizing Third Normal Form Relations from Functional Dependencies*. *ACM Trans. Database Syst.* 1(4), pp. 277–298. Available at `http://doi.acm.org/10.1145/320493.320489`.

[8] E. F. Codd (1970): *A Relational Model of Data for Large Shared Data Banks*. *Communications of the ACM* 13(6), pp. 377–387. Available at `http://doi.acm.org/10.1145/362384.362685`.

[9] E. F. Codd (1971): *Normalized Data Structure: A Brief Tutorial*. In E. F. Codd & A. L. Dean, editors: *SIGFIDET Workshop*, ACM, pp. 1–17.

[10] Bruno Courcelle (1990): *Graph Rewriting: An Algebraic and Logic Approach*. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, MIT Press, pp. 193–242.

[11] David W. Embley & W. Y. Mok (2011): *Mapping Conceptual Models to Database Schemas*. XIX, Springer, pp. 123–164. Available at `http://www.springer.com/computer/swe/book/978-3-642-15864-3`.

[12] Maribel Fernández, Hélène Kirchner & Bruno Pinaud (2017): *Strategic Port Graph Rewriting: an Interactive Modelling Framework*. Research Report, Inria ; LaBRI - Laboratoire Bordelais de Recherche en Informatique ; King's College London. Available at `https://hal.inria.fr/hal-01251871`.

[13] J. H. Jahnke & A. Zündorf (1999): *Applying Graph Transformations to Database re-engineering*. 2, World Scientific, pp. 267–286. Available at `http://www.worldscientific.com/worldscibooks/10.1142/4180`.

[14] David Maier (1980): *Minimum Covers in Relational Database Model*. *J. ACM* 27(4), pp. 664–674, doi:10.1145/322217.322223. Available at `http://doi.acm.org/10.1145/322217.322223`.

[15] David Maier, Alberto O. Mendelzon & Yehoshua Sagiv (1979): *Testing Implications of Data Dependencies*. *ACM Trans. Database Syst.* 4(4), pp. 455–469, doi:10.1145/320107.320115. Available at `http://doi.acm.org/10.1145/320107.320115`.

[16] Hossein Saiedian & Thomas Spencer (1996): *An Efficient Algorithm to Compute the Candidate Keys of a Relational Database Schema*. *Comput. J.* 39(2), pp. 124–132, doi:10.1093/comjnl/39.2.124. Available at `http://dx.doi.org/10.1093/comjnl/39.2.124`.