A Logic Framework to Support Database Refactoring

Shi-Kuo Chang[1], Vincenzo Deufemia[2], Giuseppe Polese[2], and Mario Vacca[2]

[1] University of Pittsburgh, Department of Computer Science
6101 Sennott Building, Pittsburgh, PA, USA, 15260
chang@cs.pitt.edu

[2] Università di Salerno, Dipartimento di Matematica e Informatica
Via Ponte don Melillo, 84084 Fisciano (SA), Italy
{deufemia,gpolese,mvacca}@unisa.it

Abstract. We propose a formal framework for database refactoring, analyzing both the changes to the database schema, and their impact on queries. The framework defines a logic model of changes, and views the database refactoring process as an agent based one. The agent tries to discover and resolve inconsistencies, and it is modeled as a problem solver capable to perform changes triggered upon the detection of database schema anomalies. The framework can be considered a first step towards the automation of the database refactoring process.

1 Introduction

Waterfall methodologies have their weakness in their incapability to cope with changes, which makes maintenance considerably an expensive process. For this reason, incremental and iterative methodologies were introduced [12]. They view system development as a step by step process, with the introduction of new functionalities to meet user needs. The main problem arising in both paradigms is the complexity in facing the effects of changes. Therefore, an increased automated support in this task would result in a reduction of efforts and costs, especially in incremental methodologies, because it would make them more systematic.

Changes are often necessary to reflect the continuous evolution of the real world, which causes frequent changes in functional requirements. This entails frequent modifications to the software, yielding a gradual decay of its overall quality. For this reason, many researchers in this field have developed software refactoring techniques [15]. Software refactoring is intended as the restructuring of an existing body of code, aiming to alter its internal structure without changing its external behavior [5]. It consists of a series of small behavior preserving transformations, which altogether can produce a significant software structural change. Moreover, system modifications resulting in changes to the database structure are also relatively frequent [21]. These changes are particularly critical, since they affect not only the data, but also the application programs relying on them [1,10].

Several disciplines have faced the problem of managing the effects of database schema changes. In particular, *schema modification* has faced the problem of changing the schema of a populated database. In addition to this, *schema evolution* pursues the same goal, but it tries to avoid loss of data. Alternatively, *schema versioning* performs modifications of the schema, but it keeps old versions to preserve existing queries and application programs running on it. Although schema versioning faces the problem of query and application programs preservation, it considerably increases the complexity and the overhead of the underlying DBMS. Finally, *database refactoring* aims to modify the database schema, and to change the corresponding application programs accordingly. In other words, database refactoring is the process of slowly growing a database, modifying the schema by small steps, and propagating changes to the queries [1].

So far research on database refactoring has led to the definition of several methodologies [1]. However, no significant contribution has been provided towards the automation of this process. This is mainly due to the lack of formal approaches, like those developed for schema versioning and schema evolution [3,6,11,17,18]. Nevertheless, these approaches use models that do not consider queries, hence they do not analyze the impact of schema changes on queries and application programs. In this paper we propose a formal framework for database refactoring, analyzing both the changes to the database schema and their impact on queries and application programs. The framework defines a logic model of changes, and views the database refactoring process as an agent based one. Here, the goal of agents is to discover and resolve inconsistencies. The agent is modeled as a problem solver capable to perform changes which are triggered upon the detection of database schema anomalies.

The use of elementary operators can already be found in many other approaches (see, for example, [3,6,18]), but their application relies on designer decisions or it is strongly coupled with the model features. By triggering such operators upon the detection of anomalies, our approach can potentially reduce the designer effort, providing the basis to automate the database refactoring process.

The paper is organized as follows. In section 2 we discuss related works, while section 3 introduces the approach we propose to automate database refactoring. In the sections 4 and 5 a more detailed discussion about the proposal is provided. Finally, conclusions and future works are provided in Section 6.

2 Related Work

Database refactoring is a relatively new research topic [1], and no formal approaches have been proposed for dealing with it. On the contrary, many theoretical models exist for schema evolution and schema versioning [3,6,11,17,18].

Nowadays, database researchers agree on the fact that schema evolution and versioning introduce two main problems: the *semantics of changes*, and the *change propagation.* The former requires determining the effects of changes on the schema, whereas the second analyzes the consequences of changes on data.

So far two kinds of theoretical models have been proposed: the *invariant and rule model* [3,17], and the *axiomatic or formal model* [6,18].

The *invariant and rule model* is based on the ORION object-oriented data model [3,17]. It is structured into three components: a set of properties of the schema (*invariants*), a set of schema changes, and a set of *rules*. The invariants state the properties of the schema (for example, the classes are arranged in a lattice structure), whereas rules help detecting the most meaningful way of preserving the invariants when the schema changes. This model of schema evolution yields two important issues: completeness and soundness of the schema evolution taxonomy. Both of them have been proved only for a subset of the schema change operations.

The *axiomatic model* has three basic components: *terms*, *axioms*, and *changes* [18]. The basic concept underlying this model is the *type* (analogous to the concept of class in ORION), which is in turn characterized by the *terms*. Examples of terms are the lattice of types and the set of type properties. The *axioms* state the properties of the terms, like the properties of the lattice of types. Changes on the schema are performed by means of three basic change operations: *add*, *drop*, and *modify*. The problem of the semantics of changes is solved by re-computing the entire lattice using the axioms. The model satisfies the properties of soundness and completeness.

An approach based on the axiomatic model is provided in [6], and it models schema versioning from a logical and computational point of view. In particular, it proposes a semantic and formal framework based on Description Logic [2]. The basic elements of the model are: classes and their attributes, schema (a set of class definitions), and elementary schema change operators. A basic concept underlying the framework is the *legal database instance*, which, informally represents a database instance satisfying all the constraints. This notion allows broadening the number of consistencies that are considered as reasoning problems, according to the style of Description Logic. Finally, all the consistency problems considered have been proved as decidable.

The approaches based on the two models mentioned above present two main limitations. The former regards the explosion of rules when facing more general schema changes, whereas the second regards the fact that they are all suited to the object-oriented data model. Although a taxonomy of change operations for the relational model has been proposed [20], it does not represent a complete model.

The refactoring of relational databases entails facing two important problems, which cannot be managed through the two models above: the variability of schema properties, and the propagation of changes into queries. In this paper we face both these problems.

3 Database Refactoring Through Epistemic Logic

Epistemic logic is the logic of knowledge [7,16]. It deals with the reasoning mechanisms of knowledge and with the process of *belief revision*, i.e., the evolution of a base of beliefs. In epistemic logic there are three kinds of belief changes: *expansion*, *revision*, and *contraction*. The first change refers to the addition of a belief to a base, the second is related to the addition of an inconsistent belief to a base that causes the deletion of other beliefs, and finally, the third takes into account the retraction of a belief. Epistemic logic deals with both the formulation of *postulates* for belief revision and the *constructions* of the revision process.

Database refactoring can be seen as a *revision process*. In fact, an example of schema change (together with its queries) is the *addition* of a functional dependency, which might cause the split of a table (revision) in order to keep the schema in a certain normal form. It is easy to notice that changes in a database schema depend on the properties holding in it. For instance, the addition of an attribute might only entail the modification of the table in which it is added, but it might also require more complex changes. In fact, the new attribute might alter the degree of normalization of the table if it depends only from a portion of the primary key, or it might require the introduction of new referential integrity constraints in case it coincides with the primary key of another table in the schema. Therefore, the process of database refactoring is not simply a composition of elementary changes, but it implies more sophisticated reasoning tasks, like detec! ting inconsistencies.

If we look at the schema as a knowledge base, the refactoring becomes a process of changes in the knowledge, and hence it can be interpreted as an epistemic process, which can be naturally modeled through Epistemic Logic. Within this view, it becomes natural to see refactoring as an agent managed process aiming to operate on the schema in order to perform the required changes, and trying to preserve original properties in terms of knowledge and queries.

We abide by the Thagard conception [22], which views concepts like data structures. Since a data structure can be modeled as a signature with axioms [13], we will see a database schema as a kind of data structure, and will focus on those changes involving elements in the signature (for instance, the addition or the deletion of an attribute or a functional dependency). Therefore, we need to precisely define both the knowledge on which the agent operates, and the behavior of the agent. In order to do this, we need to define

– the features of the schema;
– the allowed change requirements;
– the reasoning mechanisms of the agent.

When a change requirement arises, the agent has to decide the actions to perform. For example, when the agent receives a request of adding a new attribute, it might decide to also add one or more new functional dependencies involving the attribute. Thus, the agent is a kind of problem solver.

4 A Formalization of the Database Refactoring Problem Using Predicate Logic

In this section we formalize the problem of database refactoring using predicate logic. To this end, in the following we introduce the notations that will be used throughout the paper.

Let Σ be the set of all the attribute symbols, D the set of types, N the set of names, and V a set of variables, $A = \{(n,t)|n \in N, t \in D\}$ the set of attributes, $R = \{(n, a_1 \times \ldots \times a_m)|\ n \in N, a_1, a_2, \ldots, a_m \in \Sigma\}$ the set of relations, and $\Phi = \{(n, a_1 \times \ldots \times a_k \rightarrow b)|a_1, a_2, \ldots, a_k, b \in \Sigma, n \in N\}$ the set of functional dependencies. In order to express the schema properties, we will use the following functions and predicates: $table(R)$ to state if R is a relation, $attr(R)$ returning the set of attributes of table R. Queries are non-recursive, function-free, and Datalog formatted [4], i.e., a query is formed by a head and a body. The head is a couple $(name, X)$ with $X \in V^n$; the body is a conjunction of predicates on X. The functions $body(Q)$ and $var(Q) \subseteq V$ return the body of a query Q, and the set of its variables, respectively. Variables are labeled with the attribute to which they refer. For instance, x_a indicates that x is a variable referring to attribute a. Moreover, $FD(f)$ is a predicate that is true when f is a functional dependency, $LHS(f)$ (resp. $RHS(f)$) is a function returning the set of attributes on the left (resp. right) hand side of f, and finally, $table(f)$ returns the table to which f refers to.

Definition 1. *A database system K is a quintuple $K = (A, T, F, Q, P)$, where $A \subseteq \Sigma$, $T \subseteq R$, $F \subseteq \Phi$, Q is a set of queries, and P is a set of properties (propositions) involving elements of A, T, and F.*

Example 1. Let us consider a database system storing data about employees of a company, and having a query for retrieving all employees of the Computer Science personnel department can be represented by $K = (A, T, F, Q, P)$ where

$$\begin{aligned}
&A = \{Employee\_ID, Name, Department\_ID, Salary, Address\}\\
&T = \{R(Employee\_ID, Name, Department\_ID, Salary, Address)\}\\
&F = \{f_1 : Employee\_ID \rightarrow Name; f_2 : Employee\_ID \rightarrow Department\_ID;\\
&\qquad f_3 : Employee\_ID \rightarrow Salary, f_4 : Employee\_ID \rightarrow Address\}\\
&Q = \{q(x, y, w, z) \equiv R(x, y, \text{“}CS\text{”}, w, z)\}\\
&P = \{1) primary\_key(R, Employee\_ID)\\
&\qquad 2) \forall r \in T\ \exists k \subseteq Attr(r) \text{ such that } primary\_key(r, k)\\
&\qquad 3)\ key\_dep(r, k) \equiv \forall a \in (attr(r) - k)\\
&\qquad (\exists f \in F \text{ such that } (LHS(f) = k \wedge RHS(f) = \{a\}) \wedge\\
&\qquad (\neg \exists f \text{ such that } (LHS(f) \neq k \wedge RHS(f) = \{a\}))\\
&\qquad 4) \forall r \in T\ (primary\_key(r, k) \rightarrow key\_dep(r, k))\}
\end{aligned}$$

The properties in P state that every relation has a primary key, and the attributes fully depend on the primary key only.

Definition 2. *A database system $K = (A, T, F, Q, P)$ is said to evolve towards a database system $K' = (A', T', F', Q', P')$ iff there are four functions*
$\varepsilon_{Attr} : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$
$\varepsilon_{Table} : \mathcal{P}(R) \rightarrow \mathcal{P}(R)$
$\varepsilon_{Constr} : \mathcal{P}(\Phi) \rightarrow \mathcal{P}(\Phi)$
$\varepsilon_{P} : \mathcal{P}(Prop) \rightarrow \mathcal{P}(Prop)$
where $\mathcal{P}$ is the power set operator and $Prop$ is the set of all propositions on A, T, F;

and a substitution $\theta = (n_1 \leftarrow expr_1, \ldots, n_k \leftarrow expr_k)$ where n_j are names and $expr_j$ are expressions constituted by either single names or their conjunctions, such that
$A' = \varepsilon_{Attr}(A)$
$T' = \varepsilon_{Table}(T)$
$F' = \varepsilon_{Constr}(F)$
$P' = \varepsilon_{P}(P)$
$Q' = \{q' | \ q' = q\theta \text{ with } q \in Q\}$ *i.e., q' is obtained by applying θ to q.*

For sake of brevity, when no confusion occurs, we use symbol ε, named "evolution", to refer to the four functions together with the substitution. We also write $D' = \varepsilon(D)$.

The semantics of the database systems modeled through logic frameworks is usually specified by interpretation functions (e.g., [2]). An interpretation I is a couple $(\Delta^I, \cdot^I)$ where Δ^I is a domain and $\cdot^I$ is an interpretation function providing set theoretic interpretations. For instance, the interpretation of a relation R having two attributes is $R^I \subseteq \Delta^I \times \Delta^I$.

Given a database system K, a *database instance* on K, denoted by $\Delta(D)$, is an interpretation in K. The interpretation of a query $q \in Q$, denoted with q^I, is the set of all tuples in the database satisfying q. Two queries q and q' are equivalent in a database instance if and only if they produce the same answers. The following definition introduces the concept of query equivalence under the projection operator, which will be used for defining the concept of refactored systems.

Definition 3. *A query q is equivalent to a query q' under the projection operator π, denoted by $q \equiv_\pi q'$, if and only if*

$$\pi_{(var(q) \cap var(q'))}(q^I) = \pi_{(var(q) \cap var(q'))}(q'^I)$$

where I is an interpretation function.

Now we are ready to introduce a formal definition of *refactoring.*

Definition 4. *A database system $K = (A, T, F, Q, P)$ is said refactored in $K' = (A', T', F', Q', P')$ if and only if*

i. $\forall q \in Q \ \exists q' \in Q'$ *such that* $q \equiv_\pi q'$
ii. *if* $\forall \Delta(K) \ \Delta(K) \models P$ *then* $\forall \Delta(K') \ \Delta(K') \models P'$

Refactoring functions are particular kinds of evolution functions preserving the results of queries and the properties of the database system. For instance, if D is in third normal form, then also D' must be in the same normal form. Notice that schema evolution is a special case of refactoring. In fact, if $Q = Q'$ and $P = P'$ refactoring reduces to schema evolution.

Example 2. Let us consider the database system D introduced in example 1 and the following evolution functions:

$\varepsilon_{Attr}(A) = A$
$\varepsilon_{Table}(T) = (T - \{R\}) \cup \{R_1, R_2\}$
$\varepsilon_{Constr}(F) = F \cup \{\ f_5 :\ Department\_ID \rightarrow Address\}$
$\theta = (R \leftarrow R_1 \wedge R_2)$

where R_1 and R_2 have attributes $(Employee\_ID, Name, Salary, Department\_ID)$ and $(Department\_ID, Address)$, respectively.

By applying ε on D we obtain the database system $D' = (A', T', F', Q', P')$ with

$A' = \{Employee\_ID, Name, Department\_ID, Salary, Address\}$
$T' = \{R_1(Employee\_ID, Name, Salary, Department\_ID), R_2(Department\_ID, Address)\}$
$F' = \{f_1 : Employee\_ID \rightarrow Name; f_2 : Employee\_ID \rightarrow Department\_ID;$
$f_3 : Employee\_ID \rightarrow Salary, f_4 : Employee\_ID \rightarrow Address,$
$f_5 :\ Department\_ID \rightarrow Address\}$
$Q' = \{q(x, y, w, z) \equiv R_1(x, y, w, \text{“}CS\text{”}) \wedge R_2(\text{“}CS\text{”}, z)\}$
$P' = (P - \{primary\_key(R, Employee\_ID)\} \cup \{primary\_key(R_1, Employee\_ID),$
$primary\_key(R_2, Department\_ID)\}$

5 The Process of Database Refactoring

As the refactoring is an agent based process, in order to realize the required changes, the agent has to operate on the schema in a way that preserves the properties of the knowledge and of the queries. Two kinds of approaches can be used to accomplish this task: axiom based and constructive. The former is based on a set of postulates, known in the literature as *postulates for belief revision* [7,14]. The constructive approaches use *propositions* and *programs* for handling changes in the knowledge [8].

In the proposed refactoring process we use the constructive approach and build the evolution operator ε by using *propositions*, *questions*, and *change operations.* A question is denoted with $?p$, where p is a proposition.

An example of change operation is the splitting of a table t after the introduction of a new functional dependency f, which could be described in the following way:

$split\_table(t, t', t'', f) \leftarrow$
$(A' = A \wedge$
$T' = (T - \{t\}) \cup \{t', t''\} \wedge$

$$\begin{aligned} &F' = F \wedge \\ &Q' = \{q' \mid var(q') = var(q),\ body(q') = \rho(body(q), t, t' \wedge t'')\} \ \wedge \\ &attr(t') = attr(t) - RHS(f) \wedge \\ &attr(t'') = LHS(f) \cup RHS(f)) \end{aligned}$$

The database refactoring process is based on the following predicates: *Consistent(change-operation)*, *Hold(p)*, and *Resolve(change-operation, p)*. The former is true when the set of properties P' obtained by the application of the *change-operation* is consistent. The second is true when proposition p holds. Finally, the third is true when proposition p holds after the application of *change-operation*.

These logical operations can be expressed using the K operator of epistemic logic [9]. The K operator is applied to a proposition p using the expression Kp, whose meaning can be informally expressed by "it is known that p". As a consequence, *Hold(p)* can be expressed as Kp, *Consistent(change-operation)* as $K(\forall p \in \epsilon(P)).p$, whereas *Resolve(change-operation, p)* as Kp applied after the *change-operation*.

The agent uses the previous predicates to submit questions or to answer questions according to rules like the following:

$$\frac{\neg Consistent(change - operation)}{?\exists \omega Resolve(\omega, p)}$$

$$\frac{\neg Consistent(change - operation)}{?\exists x\ \neg Hold(x)}$$

$$\frac{\neg Hold(\neg \exists x.P(x))}{?Resolve(add(x), \exists x P(x))}$$

$$\frac{\neg Hold(\exists x.P(x))}{?Resolve(drop(x), \exists x P(x))}$$

$$\frac{?Resolve(\omega, p)}{?Consistent(change - operation)}$$

$$\frac{\neg Resolve(\omega, p)}{?\exists \omega'((\omega' \neq \omega) \wedge Resolve(\omega', p))}$$

For instance, if an inconsistency on a proposition p arises, the first rule suggests the agent to ask the question "Does there exist a change operation resolving the inconsistency?".

In general, the *reasoning process* of the agent has a question as starting point, and a change operation as ending point. The process of answering a question like the previous one is a problem solving process, since it involves the choice of a change operation. This is made through heuristics, as it usually happens in the problem solving domain [19].

Example 3. Let us consider the database system of example 1. When the agent receives a request of adding a functional dependency

$$f_5 : \; \mathit{Department\_ID} \rightarrow \mathit{Address}$$

it processes the following questions (answers are visualized in bold):

$?Consistent(add(f_5))$ **NO**
$?\exists x \; \neg Hold(x)$ **YES** $x = (LHS(f_5) \neq \mathit{Employee\_ID} \wedge RHS(f_5) = \{\mathit{Address}\})$
$?Resolve(drop(f_5), P)$ **NO**
$?Resolve(split\_table(R, R', R''), f_5)$ **YES**
$?Consistent(split\_table(R, R', R''))$ **NO**

6 Conclusions and Future Works

We have presented a formal framework for database refactoring based on epistemic logic. The framework defines a logic model of changes, and uses an agent to discover and resolve inconsistencies, and to analyze the impact of changes on queries.

In the future we would like to investigate several important issues. Firstly, it is necessary to study the system of rules and their properties. We also need the agent to be capable of making decisions. Thus, we should make the agent more autonomous and should equip it with problem solving heuristics. Moreover, we need the agent to be more communicative, in order to base its decisions also on user suggestions. For example, adding a functional dependency is a serious decision, and it would be desirable having the agent ask for user support. We would also like to investigate the possibility to exploit the second generation of epistemic logic that is based on the erotetic logic [9].

Finally, we would like to investigate the possibility of using visual language based tools capable of supporting the database refactoring process directly on the database conceptual or logic schema by means of special gesture operators.

References

1. Ambler, S.W., Sadalage, P.J.: Refactoring databases: Evolutionary database design. Addison-Wesley, London (2006)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The description logic handbook: Theory, implementation, and applications. Cambridge University Press, Cambridge (2003)
3. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, pp. 311–322. ACM Press, New York (1987)

4. Bonner, A.J.: Hypothetical Datalog: Negation and linear recursion. Rutgers University (1989)
5. Du Bois, B., Van Gorp, P., Amsel, A., Van Eetvelde, N., Stenten, H., Demeyer, S., Mens, T.: A discussion of refactoring in research and practice. Technical report, n. 2004-03, University of Antwerp, Belgium (2004)
6. Franconi, E., Grandi, F., Mandreoli, F.: A general framework for evolving schemata support. In: Proceedings of SEBD 2000, pp. 371–384 (2000)
7. Gärdenfors, P.: Belief revision: An introduction. In: Belief Revision, pp. 1–20. Cambridge University Press, Cambridge (1992)
8. Gerbrandy, J.: Dynamic epistemic logic. In: Moss, L.S., Ginzburg, J., de Rijke, M. (eds.) Logic, Language and Computation, vol. 2, pp. 67–84. CSLI Publications, Stanford (1999)
9. Hintikka, J.: A second generation epistemic logic and its general significance. In: Hendricks, et al. (eds.) Knowledge Contributors, Synthese Library no. 322, Kluwer Academic Publishers, Dordrecht (2003)
10. Karahasanovic, A.: Identifying impacts of database schema changes on application. In: Proceedings of the 8th Doctoral Consortium at the CAiSE*01, pp. 93–104 (2001)
11. Lakshmanan Laks, V.S., Sadri, F., Subramanian, I.N.: On the logical foundations of schema integration and evolution in heterogeneous database systems. In: Ceri, S., Tsur, S., Tanaka, K. (eds.) DOOD 1993. LNCS, vol. 760, pp. 81–100. Springer, Heidelberg (1993)
12. Larman, C., Basili, V.R.: Iterative and incremental development: A brief history. IEEE Computer 36(6), 47–56 (2003)
13. Luo, Z.: Program specification and data refinement in type theory. Mathematical Structures in Computer Science 3(3), 333–363 (1993)
14. Maghsoudi, S., Watson, I.: Epistemic logic and planning. In: Negoita, M.G., Howlett, R.J., Jain, L.C. (eds.) KES 2004. LNCS (LNAI), vol. 3213, pp. 36–45. Springer, Heidelberg (2004)
15. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions on Software Engineering 30(2), 126–139 (2004)
16. Meyer, J.J., Van Der Hoek, W.: Epistemic logic for AI and computer science. Cambridge University Press, Cambridge (1995)
17. Nguyen, G., Rieu, D.: Schema evolution in object-oriented database systems. Rapports de Recherche 947 (1988)
18. Peters, R.J., Özsu, M.T.: An axiomatic model of dynamic schema evolution in objectbase systems. ACM Transactions on Database Systems 22(1), 75–114 (1997)
19. Polya, G.: How to Solve It. Princeton University Press, Princeton (1957)
20. Roddick, J.F., Craske, N.G., Richards, T.J.: A taxonomy for schema versioning based on the relational and entity relationship models. In: Elmasri, R.A., Kouramajian, V., Thalheim, B. (eds.) ER 1993. LNCS, vol. 823, pp. 137–148. Springer, Heidelberg (1994)
21. Roddick, J.F.: A survey of schema versioning issues for database systems. Information and Software Technology 37(7), 383–393 (1995)
22. Thagard, P.: Conceptual revolutions. Princeton University Press, Princeton (1992)