A Conceptual Approach to Database Applications Evolution

Anthony Cleve[1], Anne-France Brogneaux[2], and Jean-Luc Hainaut[2]

[1] ADAM team, INRIA Lille-Nord Europe
Université de Lille 1, LIFL CNRS UMR 8022, France
anthony.cleve@inria.fr

[2] Faculty of Computer Science, PReCISE Research Center
University of Namur, Belgium
{afb,jlh}@info.fundp.ac.be

Abstract. Data-intensive systems are subject to continuous evolution that translates ever-changing business and technical requirements. System evolution usually constitutes a highly complex, expensive and risky process. This holds, in particular, when the evolution involves database schema changes, which in turn impact on data instances and application programs. This paper presents a comprehensive approach that supports the rapid development and the graceful evolution of data-intensive applications. The approach combines the automated derivation of a relational database from a conceptual schema, and the automated generation of a data manipulation API providing programs with a conceptual view of the relational database. The derivation of the database is achieved through a systematic transformation process, keeping track of the mapping between the successive versions of the schema. The generation of the conceptual API exploits the mapping between the conceptual and logical schemas. Database schema changes are propagated as conceptual API regeneration so that application programs are protected against changes that preserve the semantics of their view on the data. The paper describes the application of the approach to the development of an e-health system, built on a highly evolutive database.

1 Introduction

Data-intensive applications generally comprise a database and a collection of application programs in strong interaction with the former. Such applications constitute critical assets in most entreprises, since they support business activities in all production and management domains. As any software systems, they usually have a very long life, during which they are subject to continuous evolution in order to meet ever-changing business and technical requirements.

The evolution of data-intensive applications is known as a highly complex, expensive and risky process. This holds, in particular, when the evolution involves database schema changes, which in turn impact on data instances and application programs. Recent studies show, in particular, that schema evolutions may have a huge impact on the database queries occuring in the programs, reaching up to 70% query loss per new schema version [1]. Evaluating the impact of database schema changes on related programs typically requires sophisticated techniques [2,3], especially in the presence of dynamically generated queries [4]. The latter also severely complicate the adaptation of the programs to the new database schema [5,6]. Without reliable methods and tools, the evolution of data-intensive applications rapidly becomes time-consuming and error-prone.

This paper addresses the problem of automated co-evolution of database schemas and programs. Building on our previous work in the field [7,8,9], we present a comprehensive approach that supports the rapid development and the graceful evolution of data-intensive applications. The proposed approach combines the automated derivation of a relational database from a conceptual schema and the automated generation of a data manipulation API that provides programmers with a conceptual view of that relational database. This conceptual API can be re-generated in order to mitigate the impact of successive schema evolutions and to facilitate their propagation to the program level.

The remaining of the paper is structured as follows. Section 2 further analyzes the problem of data-intensive application evolutions and specifies the objective of our work. Section 3 provides a detailed presentation of our approach. A set of tools supporting the proposed approach are described in Section 4. Section 5 discusses the application of our approach and tools to a real-life data-intensive system. Concluding remarks are given in Section 6.

2 Problem Statement

2.1 Database Engineering: The Abstraction Levels

Standard database design methodologies are built on a three level architecture providing, at design time, a clean separation of objectives, and, at execution time, logical and physical independence (Figure 1). The *conceptual design* process translates users functional requirements into a *conceptual schema* describing the structure of the static objects of the domain of interest as well as the information one wants to record about them. This schema is technology-independent and is to meet semantic requirements only. The *logical design* process translates the conceptual schema into data structure specifications compliant with a database model (the *logical schema*), such as the relational, object-relational or XML models. Though it is not specific to a definite DBMS, the logical schema is, loosely speaking, platform dependent. In addition, it is semantically equivalent to the conceptual schema. The result of logical design is twofold: a logical schema and the mapping betwen the later and its source conceptual schema. This mapping defines how each conceptual object has been translated into logical constructs and, conversely, what are the source conceptual objects of each logical construct. Though mappings can be quite simple and deterministic for unsophisticate and undemanding databases, they can be quite complex for actual corporate databases. Hence the need for rigorously defined mapping. The *physical schema* specializes the logical schema for a specific DBMS. In particular, it includes indexes and storage space specifications.

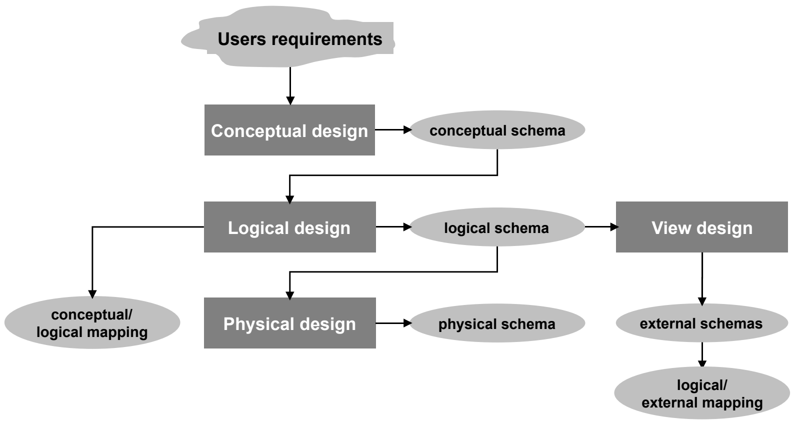


Fig. 1. Standard database design processes

Application programs interact with the database through an *external schema*, or view, generally expressed as a subset of the logical schema or as an object schema mapped on the latter. In both cases, a mapping describes how the external view has been built from the logical schema. The popular *object-relational mapping* interfaces (ORM) store this mapping explicitly, often as XML data.

2.2 Dependencies between Schemas and Programs

Current programming architectures entail a more or less strong dependency of programs on database structures, and particularly on the logical schema. Changes in the logical schema must be translated into physical schema changes and data conversion [7]. Most generally, they also lead to discrepencies between the logical and external schemas, in such a way that the latters must be adapted. In many cases, that is, when the changes cannot be absorbed by logical/external maping adaptation, the client programs must be modified as well [10]. This problem is known as a variant of the *query/view synchronization* problem [11].

As shown in [12], the use of ORM's does not solve the problem but may make things worse, since both logical and external schemas can evolve in an asynchroneous way, each at its own pace, under the responsibility of independent teams. Severe inconsistencies between system components may then progressively emerge due to undisciplined evolution processes.

In the context of this paper, we identify four representative schema modification scenarios, by focussing on the *schema* that is subject to the initial change (conceptual or logical) and its *actual* impact on client programs[1]. Let CS, LS, PS and XS denote respectively the conceptual, logical, physical and external database schemas. Let $\mathcal{M}(CS, LS)$ and $\mathcal{M}(LS, XS)$ denote the conceptual/logical and logical/external mappings, and let P be a program using XS.

[1] Existing taxonomies for schema evolution [13,14] usually consider the *reversibility* of the schema change and its *potential* impact on the programs.

1. **Change in CS without impact on P.** The conceptual schema CS undergoes structural modifications that translate domain evolution (we ignore refactoring changes). However, the modified conceptual constructs are outside the scope of the part of external schema XS that is used by program P. Conceptually, this modification must be followed by the replay of the *logical design* and *view design* processes, which yields new versions of logical schema LS and mappings $\mathcal{M}(CS, LS)$ and $\mathcal{M}(LS, XS)$. Theoretically, this evolution should have no impact on P. However, the extent to which this property is achieved depends on two (1) the quality of the *logical design* process and (2) the power of the *logical/external mapping* technology and the care with which this mapping is adapted, be it automated or manual.
2. **Change in CS with impact on P.** In this scenario, conceptual changes cause the modification of the part of external schema XS used by P, in such a way that no modification of mapping $\mathcal{M}(LS, XS)$ is able to isolate program P from these conceptual changes. Accordingly, some parts of P must be changed manually. Considering the current state of the art, the best one can expect from existing approaches is the identification of those parts.
3. **Change in LS with CS unchanged.** The logical schema LS is refactored to better suit non functional requirements such as adaptation to another platform, performance, structural quality, *etc.* As in scenario 1, this modification requires replaying the *logical design* process and yields new versions of LS, $\mathcal{M}(CS, LS)$ and $\mathcal{M}(LS, XS)$. Automatic program adaptation has proved tractable through *co-transformation* techniques that couple schema and program transformations [15,9,6].
4. **Change in PS with CS and LS unchanged.** Such a modification is fully controlled by the DBMS and has no impact on program P, which, at worst, could require recompilation.

Our proposal aims to address the evolution problems posed by scenarios 1, 2 and 3 via a tool-supported programming architecture that allows programs to

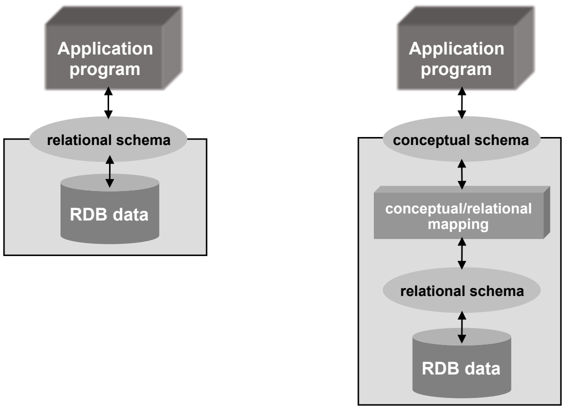


Fig. 2. Programs interact with a logical (left) or a conceptual (right) database

interface with the database through an external conceptual view instead of a logical view (Figure 2). This architecture provides logical independence in scenarios 1 and 3 by encapsulating mappings $\mathcal{M}(CS, LS)$ and $\mathcal{M}(LS, XS)$ into an API automatically generated by an extension of the DB-MAIN CASE tool. In scenario 2, the identification of the program parts that must be changed is carried out by the recompilation process driven by the strong typing of the API. Though seeking full automation would be irrealistic, the approach provides a reliable and low cost solution that does not require high skills in system evolution.

3 Approach

The approach proposed in this paper relies on a generic representation of schemas, and on a formal definition of schema transformations and schema mappings. It consists in automatically building a schema mapping during the logical design process, in order to use such a mapping as input for (re)generating a conceptual data manipulation API to the relational database.

3.1 Generic Schema Representation

The approach deals with schemas at different levels of abstraction (conceptual, logical, physical) and relying on various data modeling paradigms. To cope with this variety of models, we use a large-spectrum model, namely the Generic Entity-relationship model [16], or GER for short. The GER model is an extended ER model that includes most concepts of the models used in database engineering processes, encompassing the three main levels of abstractions, namely conceptual, logical and physical. In particular, it serves as a generic pivot model between the major database paradigms including ER, relational, object-oriented, object-relational, files structures, network, hierarchical and XML. The GER model has ben given a formal semantics [16].

3.2 Transformational Schema Derivation

Most database engineering processes can be modeled as chains of schema transformations (such a chain is called a *transformation plan*). A schema transformation basically is a rewriting rule T that replaces a schema construct C by another one $C' = T(C)$. A transformation that preserves the information capacity of the source construct is said *semantics-preserving*. This property has been formally defined and its applications have been studied in [16]. In the context of system evolution, schema transformations must also specify how instances of source construct C are transformed into instances of C'.

The *logical design* process can be described as follows: we replace, in a conceptual schema, the ER constructs that do not comply with the relational model by application of appropriate schema transformations. We have shown in [16] that (1) deriving a logical (relational) schema from an ER schema requires a dozen semantics-preserving transformations only and that (2) a transformation plan made up of semantics-preserving transformations fully preserves the semantics of the conceptual schema. The transformations on which a simple relational logical design process relies includes the following : transforming is-a relations into one-to-one rel-types, transforming complex rel-types into entity types and functional (one-to-many) rel-types, transforming complex attributes into entity types, transforming functional rel-types into foreign keys. A transformation plan should be idempotent, that is, applying it several times to a source schema yields the same target schema as if it was applied only once.

The transformational approach to database engineering provides a rigorous means to specify forward and backward mappings between source and target schemas: the trace (or history) of the transformations used to produce the target schema. However, as shown in [7], extracting an synthetic mapping from this history has proved fairly complex in practice. Therefore, we will define in the next sections a simpler technique to define inter-schema mappings.

3.3 Schema Mapping Definition

Let $\mathcal{C}$ be the set of GER schema constructs. Let $\mathcal{S}$ be the set of GER schemas.

Definition 1 (Schema). *A schema $S \in \mathcal{S}$ is a non empty set of schema constructs : $S = \{C_1, C_2, ..., C_n\} : \forall i : 1 <= i <= n : C_i \in \mathcal{C}$.*

We assume that each schema construct $C \in \mathcal{C}$ belongs to at most one schema : $\forall S_1, S_2 \in \mathcal{S} : S_1 \neq S_2 : S_1 \cap S_2 = \emptyset$.

Definition 2 (Construct mapping $\mathcal{M}_c$). *A construct mapping $\mathcal{M}_c = (\mathcal{V}, \mathcal{E})$ is an hypergraph consisting of vertices $\mathcal{V} = \{C_i \in \mathcal{C}\}$ representing schema constructs, and hyperedges $\mathcal{E} = \{e_j \in 2^{\mathcal{V}}\}$, each representing a mapping relationship between the schema constructs it connects.*

Definition 3 (Schema mapping $\mathcal{M}_s$). *A schema mapping $\mathcal{M}_s(S_1, S_2) = (\mathcal{V}, \mathcal{E})$ between two schema $S_1, S_2 \in \mathcal{S}$ is a sub-hypergraph of $\mathcal{M}_c$ where:*

- *$\mathcal{V} \subseteq S_1 \cup S_2$, each vertice corresponds to either a construct of S_1 or of S_2;*
- *$\forall e \in \mathcal{E} : e \cap S_1 \neq \emptyset \wedge e \cap S_2 \neq \emptyset$, each hyperedge connects at least one construct of S_1 to at least one construct of S_2.*

Note that the above definition of schema mapping is very generic. First, it allows *many-to-many* and *multi-schema* mappings to be defined. Second, it does not make any further assumptions regarding the *meaning* of the mapping relationships. In this paper, the schema mapping represents an *ancestor-child* relationship between conceptual and relational constructs.

Definition 4 (Schema mapping transitivity). *A set of schema mappings defined over a set of schemas $\Gamma \subseteq 2^{\mathcal{S}}$ is transitive when $\forall S_1, S_2, S_3 \in \Gamma$: $(\exists e_1 \in edges(M_s(S_1, S_2)) \wedge \exists e_2 \in edges(M_s(S_2, S_3)) \wedge e_1 \cap e_2 \neq \emptyset) \implies \exists e_3 \in edges(M_s(S_1, S_3)) : e_3 = (e_1 \cap S_1) \cup (e_2 \cap S_3)$.*

When schema mapping transitivity does not hold explicitly, one may need to derive indirect mappings through *schema mapping composition* [17]. Starting from a schema mapping over schemas S_1 and S_2 ($\mathcal{M}_s(S_1, S_2)$) and a mapping over schemas S_2 and S_3 ($\mathcal{M}_s(S_2, S_3)$) , the goal of such a composition is to derive an equivalent mapping over S_1 and S_3 ($\mathcal{M}_s(S_1, S_3)$). In this paper, mapping transitivity is garanteed through a mapping propagation technique described in the next section.

3.4 Schema Mapping Propagation

Considering that a transformation process involves a chain of successive versions of a schema $S_1, S_2, \cdots, S_n$, the goal of mapping propagation is to guarantee that, at the end of the process, the mapping between the source and target schemas $\mathcal{M}_s(S_1, S_n)$ corresponds to the composition of the mappings between the successive schema versions $\mathcal{M}_s(S_1, S_2), \ldots, \mathcal{M}_s(S_{n-1}, S_n)$. In other words, after k transformations applied to the source schema S_1, the set of schema mappings defined over the set of schemas $\{S_1, S_2, \cdots, S_{k+1}\}$ must be transitive.

As defined above, a schema mapping is represented through a static *many-to-many* relationship from the constructs of one of the schemas to those of the other. In our approach, each source object includes a (set of) mapping identifier(s) (also called a *stamp*) that is transmitted to all the target objects that derive, directly or indirectly, from it. Suppose that we have a source schema S_1 and a target schema S_2. S_2 results from the transformation of S_1 according to a given transformation process T: $S_2 = T(S_1)$. The mapping propagation process then relies on the following principles:

- Before executing the transformation process (T), each object of S_1 receives a proper mapping identifier, that is, a stamp that is not inherited.
- The transformations applied on objects of S_1 propagate automatically the mapping information to the resulting objects of S_2 in the form of inherited mapping identifier. It is a copy of the proper mapping identifier if the object of S_1 has a proper stamp only.
- If the source object has received inherited stamps through a previous transformation, then the inherited stamps are copied instead.
- If one object is transformed into several objects (as is the case for the instantiation transformation of a multivalued attribute into a list of single-valued attributes), the target objects all get the same inherited stamp.
- If several objects are transformed into one object, the new object inherits the list of the proper or inherited stamps of all the source objects.

By examining the S_1 and S_2 objects that share the same mapping identifiers, an analyzer can infer which transformations have been applied. Mapping propagation and analysis must help database engineers to achieve such tasks as evolution, migration, view derivation and integration. In this paper, the propagated schema mapping serves as a basis for the automatic generation of a conceptual database manipulation API.

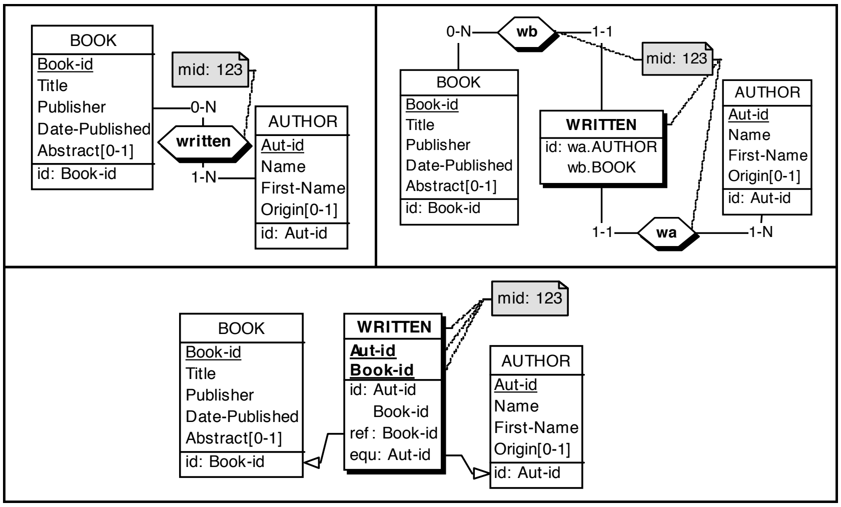


Fig. 3. Propagation of mapping identifiers through two successive transformations

Figure 3 illustrates the mapping propagation process. In this case, the proper mapping identifier (*mid: 123*) of rel-type `written` is propagated through two successive transformations. In the first step, the *many-to-many* rel-type `written` between `AUTHOR` and `BOOK` is transformed into an entity type `WRITTEN`. This semantics-preserving transformation creates also two *one-to-many* rel-types (`wa` between `AUTHOR` and `WRITTEN`, `wb` between `BOOK` and `WRITTEN`). The initial mapping identifier of rel-type `written` is inherited by entity type `WRITTEN` and rel-types `wa` and `wb` (*mid':123*). In the second step, the *one-to-many* rel-types `wa` and `wb` are transformed into referential constraint (concept that represents *foreign keys* in relational model). This transformation preserves also the semantic of original schema. The inherited mapping identifiers of *one-to-many* rel-types are propagated to the equivalent foreign keys `Aut-id` and `Book-id` of `WRITTEN`.

3.5 API Generation

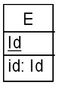
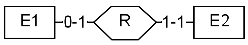
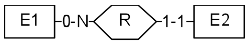
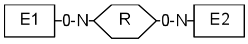
The formal mapping between the conceptual and relational schema allows the automatic generation of a conceptual data manipulation API. The generated API provides a natural and platform-independent view of the relational database. In this section, we present the principles on which the API generation process relies.

Class hierarchy. The generated API aims at providing application programs with a conceptual view of the relational database. Consequently, the mapping between the conceptual schema and the class hierarchy of the API is straithforward. Each entity type E belonging to the conceptual schema corresponds to a public Java class E in the generated API. Each attribute A of E corresponds to an instance variable A of class E. Similarly, each *at-most-one* role played by E translates into one instance variable of the class associated with E.

In the presence of *is-a* relationships (e.g., E_2 *is-a* E_1), an equivalent class hierarchy is produced (e.g., `public class` E_2 `extends` E_1). In this paper, we assume that each *is-a* relationship of the conceptual schema corresponds to a *partition*. That is, the set of supertype instances is equivalent to the union of the sets of the sub-type instances (totality), and the sets of sub-type instances are disjoint.

Data navigation. The API offers two main ways of navigating the data: by entity type and by relationship type. The corresponding methods are summarized in Table 1. Since we only consider partition-based isa-hierarchies, all the objects returned by the API to the client applications are *full* objects that are constructed from a particular leaf of the class hierarchy. This allows the client applications to down-cast the resulting object to a certain sub-level class whenever needed.

Table 1. Data navigation through the conceptual API

Navigation type	Method signature	Result description
By entity type 	`Vector<`E`> getAll`E`()`	all the instances of entity type E
	E `get`E`ById(`$IdType\ Id$`)`	the instance of entity type E having Id as identifier value
By one-to-one relationship type 	E_2 E_1`.get`E_2`Via`R`()`	the instance of E_2 associated with a given instance of E_1 through R
	E_1 E_2`.get`E_1`Via`R`()`	the instance of E_1 associated with a given instance of E_2 through R
By one-to-many relationship type 	E_1 E_2`.get`E_1`Via`R`()`	the instance of E_1 associated with a given instance of E_2 through R
	`Vector<`E_2`>` E_1`.getAll`E_2`Via`R`()`	all the instances of E_2 associated with a given instance of E_1 through R
By many-to-many relationship type 	`Vector<`E_1`>` E_2`.getAll`E_1`Via`R`()`	all the instances of E_1 associated with a given instance of E_2 through R
	`Vector<`E_2`>` E_1`.getAll`E_2`Via`R`()`	all the instances of E_2 associated with a given instance of E_1 through R

Data modification. Each generated class E also provides the client applications with data modification methods, allowing (1) to insert a new instance of E in the database, (2) to update the value of each conceptual attribute of a given instance of E, (3) to connect/disconnect an instance of E according to each relationship type in which E may play a role, and (4) to delete an instance of E.

The insertion of an entity type instance is implemented in two steps. First, the object is created via its constructor, which may necessitate the recursive invocation of supertype constructors. Second, the internal `store` method is called. This method allows the insertion of a new row in each table associated with the entity type hierarchy, starting from its root (recursive call to the `store` method of the supertype). The arguments of the `insert` method for a given entity type E consist of all the (inherited) conceptual attributes of E and all the mandatory `at-most-one` roles (indirectly) played by E. The method returns an object of class E that corresponds to the inserted instance.

Updating an attribute value is implemented as a simple `update` query. The value of optional attributes can be set to `null`. The same holds for each *at-most-one* role that can be played by the associated entity type: disconnection is only allowed if the role is optional.

The *delete* method is in charge of discarding a particular entity type instance e from the database. The delete process, which relies on the *delete cascade* mechanism, involves two steps. First, all the entity type instances associated to e are either (recursively) deleted or disconnected (depending on the relationship cardinalities). Second, the table row(s) representing e are removed from the database, starting from the leafs of the hierarchy.

3.6 Schema Evolution Revisited

The approach proposed in this paper has been considered so far as a means to rapidly develop data-intensive applications. Let us now further examine the advantages of this approach in the context of *schema evolution*, by revisiting the three schema evolution scenarios identified in Section 2 as problematic:

1. **Change in CS without impact on P.** Our approach to logical design based on an idempotent schema transformation plan allows the automatic derivation of a new logical schema LS' from the new CS', and the automatic adaptation of the mapping between them. Although the existing programs are not impacted by the schema evolution, regenerating the conceptual API allows one to immediatly replay the view design process such that new client programs can be more easily developed.
2. **Change in CS with impact on P.** In this second scenario, the conceptual modifications concern a schema part that is used by the existing programs. The new conceptual schema CS' must then be translated in a new relational schema LS'. Without our conceptual API, this would necessitate to (manually) rewrite all the SQL queries that access the tables of LS subject to evolution in LS'. According to our approach, this task can be done automatically by simply regenerating the conceptual API. However, the client applications still need to be adapted to the new API, but the source code locations where adaptation should take place can be statically detected by the Java compiler. This would not have been the case with client applications directly accessing the relational database by means of dynamically generated SQL queries.

3. **Change in LS with CS unchanged.** This scenario only requires the re-generation of the conceptual API, and is fully transparent for the client applications. Indeed, refactorings applied to the logical schema LS theoretically allow the $\mathcal{M}_s(CS, LS)$ mapping to be automatically adapted, and thus the conceptual API to be regenerated with no impact on the client programs (the API method signatures remain unchanged).

4 Tool Support

The approach presented in this paper is supported by a set of tools developed in DB-MAIN [18], a generic modeling tool dedicated to database applications engineering, and in particular to database design, reverse engineering, re-engineering, integration, maintenance and evolution. DB-MAIN offers general functions and components that allow the development of sophisticated processors supporting the mapping management. It provides the user with a toolbox of more than 25 schema transformations sufficiently rich to encompass most database engineering processes. These transformations are designed to propagate mapping identifiers from source to target objects and, therefore, to automatically construct the mapping between different schemas.

The conceptual API generator is implemented as a Java plugin of DB-MAIN, that consists of about 3500 lines of code. The generator takes as inputs (1) a conceptual schema, (2) a logical relational schema, and (3) the mapping between the two schemas. It generates as output a set of dedicated Java classes allowing client applications to manipulate the relational database in a conceptual and transparent manner.

5 Case Study

The tool-supported approach presented in this paper has been recently applied to the development of the Gisele system. Gisele is a project devoted to the modeling and validation of clinical pathways [19]. The database of the system stores pathway models, their successive versions and revisions, their executions (*cases*), resources description, access control models, organizational and information entities description, goals and validation procedures and results. This database being in constant evolution, special attention has been paid to minimizing the impact of schema evolution on client programs.

Table 2 provides some statistics about both the conceptual and logical schema of the Gisele system. The conceptual schema is made up of 49 entity types, 60 attributes and 63 relationship types (mostly *one-to-many*). Some entity types are organized in *is-a* hierarchies of a maximal depth of 4. The logical relational schema was automatically derived from the conceptual schema using our transformation plan for relational schema design [16]. The resulting logical schema contains 58 tables, 207 columns and 107 foreign keys. Those statistics reveal that the mapping $\mathcal{M}_s(CS, LS)$ that holds between the conceptual and the logical schema is quite complex in this case study.

Table 2. Schema statistics of the Gisele system

Schema construct	**Conceptual schema**	**Logical schema**
# of entity types	49	58
# of *is-a* relationships	10	-
# of attributes	60	207
# of relationship types	63	-
# of *one-to-one* relationship types	1	-
# of *one-to-many* relationship types	53	-
# of *many-to-many* relationship types	9	-
# of foreign keys	-	107

Starting from those two schemas and the mapping between them, our generator produced a comprehensive API made up of 50 java classes, and totalling more than 20 thousands lines of code. This API has been systematically and successfully tested by means of several client applications, which in turn were automatically generated from the conceptual schema.

Since its initial version in October 2009, the conceptual schema of Gisele has been subject to multiple evolutions. Indeed, this schema is composed of a quite stable kernel, on top of which additional dimensions have been successively introduced. The kernel models the main standard concepts of workflow specifications, like processes, sequence flows, message flows, etc. The additional dimensions introduce such notions as time, resources and evolution.

The API generator proved very convenient when adapting existing Gisele applications to the successive versions of the conceptual schema. A large part of the evolutions were additive, and therefore, had no impact at all on the client programs. Some schema evolutions caused the mapping between the conceptual and logical schemas to be changed. Regenerating the API allowed us to make such evolutions fully transparent for the existing client applications. Finally, a few conceptual modifications actually impacted on the client programs (updated attribute/names, new mandatory attributes/roles). In the latter case, the presence of the conceptual API allowed the Java compiler to statically detect inconsistencies between client applications and the new conceptual schema.

6 Conclusions

This paper has briefly described a system architecture the goal of which is to ease the development and maintenance of evolving data-intensive applications. Though the Gisele experiment is not quite complete at the present time, we can already draw useful conclusions.

- Programming with the conceptual API proves easier than with JDBC. In particular, based on a static query interface, the source code is more expressive and easier to maintain. In addition, it allows static error detection (in

particularly inconsistencies between logical and external schemas) at compile time. This advantage is not surprising, since it is one of the motivations of such languages as SQLJ [20] (SUN) and LINQ [21] (Microsoft).
- The architecture offers the same advantages as data wrappers [8]. For instance, the code of the API methods can include code devoted to additional services such as access control or statistics collection.
- The approach is not limited to relational DBMS. It can be used to integrating other data managers into a unique abstract data model.
- As compared with current Object/Relational Mapping (ORM) middlewares, the proposed approach automatically maintains the consistency between the logical (e.g., relational) and external schemas.
- Due to the simplicity of the data model API, building higher-level abstract objects on the conceptual schema is both easy and reliable. For instance, in the Gisele system, we have defined objects PATHWAY and CASE on top of the Gisele API.

We anticipate two main directions for future work on this topic. First, we intend to further evaluate the usability and scalability of our approach based on additional real-life experiments. Second, we would like to explore the extension of our methodology to other evolution scenarios as schema integration or database platform migration.

Acknowledgments. Anthony Cleve is supported by an ERCIM "*Alain Bensoussan*" Fellowship Programme. Partial support was received from the Belgian *Région Wallonne* (GISELE project), from the IAP Programme of the Belgian State, Belgian Science Policy (MoVES project), and from the Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER (CPER 2007-2013).

References

1. Curino, C.A., Moon, H.J., Tanca, L., Zaniolo, C.: Schema evolution in wikipedia: toward a web information system benchmark. In: International Conference on Enterprise Information Systems (ICEIS), pp. 323–332 (2008)
2. Karahasanovic, A.: Supporting Application Consistency in Evolving Object-Oriented Systems by Impact Analysis and Visualisation. PhD thesis, University of Oslo (2002)
3. Maule, A., Emmerich, W., Rosenblum, D.S.: Impact analysis of database schema changes. In: Proceedings of the 30th international conference on Software engineering (ICSE 2008), pp. 451–460. ACM Press, New York (2008)
4. Cleve, A., Hainaut, J.L.: Dynamic analysis of sql statements for data-intensive applications reverse engineering. In: Proceedings of the 15th Working Conference on Reverse Engineering, pp. 192–196. IEEE, Los Alamitos (2008)
5. Curino, C., Moon, H.J., Zaniolo, C.: Graceful database schema evolution: the prism workbench. Proceedings of the VLDB Endowment 1(1), 761–772 (2008)
6. Cleve, A.: Program Analysis and Transformation for Data-Intensive System Evolution. PhD thesis, University of Namur (October 2009)

7. Hick, J.M., Hainaut, J.L.: Database application evolution: A transformational approach. Data & Knowledge Engineering 59, 534–558 (2006)
8. Thiran, P., Hainaut, J.L., Houben, G.J., Benslimane, D.: Wrapper-based evolution of legacy information systems. ACM Trans. Software Engineering and Methodology 15(4), 329–359 (2006)
9. Cleve, A., Hainaut, J.L.: Co-transformations in database applications evolution. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 409–421. Springer, Heidelberg (2006)
10. Hainaut, J.L., Cleve, A., Henrard, J., Hick, J.M.: Migration of legacy information systems. In: Software Evolution, pp. 105–138. Springer, Heidelberg (2008)
11. Rundensteiner, E.A., Lee, A.J., Nica, A.: On preserving views in evolving environments. In: KRDB. CEUR Workshop Proceedings, CEUR-WS.org, vol. 8, pp. 13.1–13.11 (1997)
12. Hainaut, J.L.: Legacy and future of data reverse engineering. In: Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009), p. 4. IEEE Computer Society, Los Alamitos (2009)
13. Shneiderman, B., Thomas, G.: An architecture for automatic relational database sytem conversion. ACM Trans. Database Syst. 7(2), 235–257 (1982)
14. 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)
15. Visser, J.: Coupled transformation of schemas, documents, queries, and constraints. Electronic Notes in Theoretical Computer Science 200(3), 3–23 (2008); Proceedings of the 3rd International Workshop on Automated Specification and Verification of Web Systems (WWV 2007)
16. Hainaut, J.L.: The transformational approach to database engineering. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 95–143. Springer, Heidelberg (2006)
17. Bernstein, P.A., Green, T.J., Melnik, S., Nash, A.: Implementing mapping composition. In: VLDB 2006: Proceedings of the 32nd international conference on Very large data bases, VLDB Endowment, pp. 55–66 (2006)
18. DB-MAIN: The DB-MAIN official website (2010), `http://www.db-main.be`
19. Damas, C., Lambeau, B., Roucoux, F., van Lamsweerde, A.: Analyzing critical process models through behavior model synthesis. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), pp. 441–451. IEEE, Los Alamitos (2009)
20. Clossman, G., Shaw, P., Hapner, M., Klein, J., Pledereder, R., Becker, B.: Java and relational databases: SQLJ (tutorial). In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (1998)
21. Meijer, E., Beckman, B., Bierman, G.M.: Linq: reconciling object, relations and xml in the .net framework. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, p. 706. ACM, New York (2006)