Reverse Engineering User Interfaces for Interactive Database Conceptual Analysis

Ravi Ramdoyal[1], Anthony Cleve[2], and Jean-Luc Hainaut[1]

[1] Laboratory of Database Application Engineering - PReCISE Research Center
Faculty of Computer Science, University of Namur, Belgium
{rra,jlh}@info.fundp.ac.be
[2] INRIA Lille-Nord Europe, LIFL CNRS UMR 8022
University of Lille 1, France
anthony.cleve@inria.fr

Abstract. The first step of most database design methodologies consists in eliciting part of the user requirements from various sources such as user interviews and corporate documents. These requirements formalize into a conceptual schema of the application domain, that has proved to be difficult to validate, especially since the visual representation of the ER model has shown understandability limitations from the end-users standpoint. In contrast, we claim that prototypical user interfaces can be used as a two-way channel to efficiently express, capture and validate data requirements. Considering these interfaces as a possibly populated physical view on the database to be developed, reverse engineering techniques can be applied to derive their underlying conceptual schema. We present an interactive tool-supported approach to derive data requirements from user interfaces. This approach, based on an intensive user involvement, addresses a significant subset of data requirements, especially when combined with other requirement elicitation techniques.

Keywords: Information systems engineering, Requirements engineering, Database engineering, Human-computer interfaces reverse engineering.

1 Introduction

Data modeling plays a pivotal role in Requirements engineering, as it defines the semantic core of the future application. Accurately eliciting and validating user requirements are vital to build reliable specifications of the data application domain. Database engineering precisely focuses on data modeling, where these requirements are typically expressed by means of a conceptual schema, which is an abstract view of the static objects of the application domain. Designing databases often relies on various requirements elicitation techniques such as the analysis of corporate documents and interviews of stakeholders. However beyond the initial collection of the requirements, these techniques usually do not actively and interactively involve end-users.

B. Pernici (Ed.): CAiSE 2010, LNCS 6051, pp. 332–347, 2010.

Still, the necessity to associate end-users of the future system with its specification and development steps has long been advocated [1]. End-users can perceive the qualities and the flaws of the information systems currently used, and since they know "how business is done" in the developing environment, they have the ability to state what could be done to improve it [2]. Involving them in the expression of their needs and in the definition of an appropriate solution can therefore reduce their resistance toward a new information system infrastructure and stimulate productivity [3]. As for the validation of these data requirements, their formal graphical representation is often difficult to grasp for the end-users. Indeed, while analysts focus on building requirements meeting various expectations (correctness, completeness, consistency, ...), requirements from the end-users standpoint need to be understandable and expressive enough.

In order to tackle this issue, we present an approach to elicit and validate database requirements, based on end-users involvement through interactive prototyping, and adapting techniques coming from various fields of study. By taking advantage of their expressiveness and understandability, we adopt form-based user interfaces as a two-way channel to efficiently express, capture and validate data requirements with end-users. In particular, we capitalize on the transformational power of data structure Reverse engineering techniques, which aim at extracting specifications from existing artifacts.

This approach relies on the principles of the ReQuest framework [4,5], which provides a complete methodology and a set of tools to deal with the analysis, development and maintenance of web applications. This approach proved that it is possible to efficiently and swiftly involve end-users in the definition of their needs. Whereas ReQuest deals with data modeling and the dynamic aspects of the future application (such as task analysis, behavior of the application, etc.), while providing generators for several of its components (database, framework skeleton, etc.), here we focus specifically on improving the data requirements process, leading the interfaces to appear as a means rather than an end product.

The remainder of the paper is structured as follows. Section 2 delineates the research context, while Section 3 describes the related works. The main principles of the proposal are detailed in Section 4. In Section 5, we elaborate on key aspects of our approach, namely Reverse engineering, modular refinement, view integration and transformational approach. Section 6 briefly presents the tool kit supporting the proposed methodology. Finally, in Section 7, we discuss the merits and limitations of our proposal and anticipate future work.

2 Research Context

The process of designing and implementing a database that has to meet specific user requirements has been described extensively in the literature [6] and has been available for several decades in CASE tools. It consists of four main subprocesses: (1) *Conceptual design* through which user requirements are translated into a conceptual schema; (2) *Logical design*, which produces an operational logical schema that translates the constructs of the conceptual schema according

to a specific technology family; (3) *Physical design*, which augments the logical schema with performance-oriented DBMS-specific constructs and parameters; (4) *Coding*, which translates the physical schema (and some other artifacts) into the DDL (*Data Definition Language*). Transformational and generative techniques allow one to automate the production of logical and physical counterparts of the conceptual schema [7], as well as artifacts of the final application [8], such as database code, program fragments, interface forms, etc.

In this paper, we focus on the primordial conceptual design, for which the Entity-Relationship (ER) model has long been the most popular medium to express conceptual requirements [9]. Its simplicity, its graphical representation, the availability of numerous CASE tools that include an ER schema editor (should) make it the ideal communication medium between designers and users. However, despite its merits, the ER formalism often fails to meet its objectives as an effective end-users communication medium. The reason is easy to grasp: a conceptual ER schema is actually a graphical presentation of a large and complex set of 1st and 2nd order predicates, and implicitly conveys non trivial concepts such as sets, non-1st normal form relations (NF^2), algebraic operators, candidate keys and functional dependencies. Fig. 1(a) shows a small conceptual schema and its NF^2 relational interpretation according to the GER formalism [7]. The intrinsic complexity of the requirements has been concealed by the apparent intuitiveness of the ER graphical notation but has not disappeared.

On the other hand, most users are quite able to deal with complex data structures, provided they are organized according to familiar layouts. In particular, electronic forms have proved to be more natural and intuitive than usual conceptual formalisms to express data requirements [10], while making the semantics

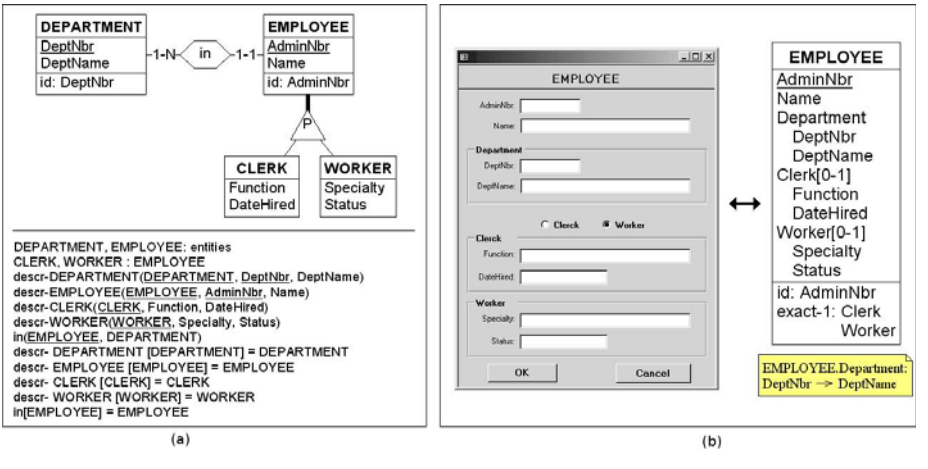

Fig. 1. (a) An ER schema and its formal expression. (b) An almost equivalent electronic form and its informational contents.

of the underlying data understandable [11]. This naturally leads to ponder the idea of using such forms as the preferred way to describe data structures in requirements engineering processes.

3 State of the Art

The strong link existing between graphical interfaces and data models is usually exploited in forward engineering. Indeed, as we have seen, it is relatively straightforward to produce artifacts such as form-based interfaces from a conceptual schema. Conversely, a form contains data structures that can be seen as a particular *view* of the conceptual schema. The transition from one to another has been shown to be tractable [12], so that database Reverse engineering techniques can be applied to recover a fragment of the conceptual schema. These techniques can be combined with prototyping, which may also act as a basis for interviews or group elicitation [13], while providing early feedback [14].

Deriving requirements from prototype artifacts has a long tradition. In 1984, Batini et al. studied paper forms as a widely used means to collect and communicate data in the office environment. Later on, Choobineh et al. [10] explored a form-based approach for database analysis and design, and developed an analyst-oriented Form Definition System and an Expert Database Design System that incrementally produced an ER diagram based on the analysis of a set of forms. Kösters et al. [15] introduced a requirements analysis method combining user interface and domain analysis. Rollinson and Roberts studied the problem of non-expert customization of database user interfaces in [12], and developed a set of graph-oriented transformations to extract an Extended Entity-Relationship schema describing an interface's information content. More recently, Yang et al. inquired about the WYSIWYG user-driven development of Data Driven Web Applications, while transparently generating their underlying application model on the fly [16]. We can observe that all these approaches rely on the same core principles: (1) build a set of form-based interfaces; (2) extract the underlying form model; (3) translate the form model into a working data model; (4) progressively build an integrated data model by looking for structural redundancies as well as constraints and dependencies.

We also notice that the number of studies on the subject is limited (especially recently), and that several limitations must be underlined in most of them. First of all, the tools provided for the drawing of the interfaces are not dedicated to this purpose and/or not convenient for end-users. Secondly, the underlying form model of the interfaces must often be constructed by analyzing the physical composition (layout) before the informational composition (content) of the form, and in parallel, the prototypical form-based interfaces do not use a generic language that would enable GUI generation of an application on any target platform. Regarding the coherence of the interfaces, it is assumed that the labels are used consistently through out the different forms, and little care is given to possible lexical variation (paronymy, feminine, plural, spelling, mistakes, etc.) and ontological ambiguity (polysemy, homography, synonymy). The use of examples

(either through static statements or dynamic interaction) is not systematically used to elicit constraints and dependencies. And last but not least, the final output is limited to the prototype itself and the final version of underlying integrated data model is not systematically submitted to the end-users in a way enabling easy validation.

4 Proposal

To alleviate understandability limitations of the ER model, we propose to use form-based user interfaces as a two-way channel to efficiently capture and validate static data requirements with end-users by providing the latter with adequate techniques to draw the interfaces describing the underlying key concepts of their application domain. This approach benefits from the advantages of rapid prototyping (such as visual expressiveness, early feedback and clarification) [17], while making the user a central actor of the process.

Our RAINBOW approach involves end-users in a simple and interactive way while providing the analysts with semi-automatic tools. The approach is formalized into a seven step process whose aim is to support the development of future applications and answer most of the concerns raised in Section 3. To illustrate these steps, let us consider the example of a small sales company wanting to develop a simple IT solution to manage its customers and their orders. Each order is created in a given shop and specifies a list of products.

1. Represent: End-users are invited to draw a set of form-based interfaces to perform usual tasks of their application domain. Such interfaces are typically entry forms to capture data on, say, a new customer or a new product. The end-users must at least provide basic properties regarding the interface elements (typically a label and description). Advanced users may also provide other properties such as the size of a field, the expected type of values, default or predefined values, existence constraints, as well as links between the concepts. Note that the objective is *not* to let end-users draw the interfaces of a future application[1], but to capture requirements through a medium they are familiar with. A dedicated drawing tool provides them with a limited set of primitive widgets, namely *interfaces, group boxes, tables, input fields, selection fields* and *button panels* (Fig. 2). These simple but usual form widgets can indeed be used to express any other complex widget. Fig. 3(a) illustrates the key concepts (`Customer, Order, Product, Shop`) that the end-users might draw for the running example.

2. Adapt: Once the interfaces are drawn, *database Reverse engineering techniques* are applied to recover the underlying conceptual schema of the domain. The interfaces (Fig. 3(a)) are automatically analyzed to extract data models using mapping rules, a subset of which is presented in Fig. 2. The stereotype `<R>` indicates an attribute associated to a button panel. Then, each individual entity type (Fig. 3(b)) is transformed into a primitive conceptual schema by transforming complex attributes (Fig. 3(c)). The structure of the data models is very simple, but, as we will discuss it, there is no semantic loss.

[1] Which is the case in the ReQuest framework.

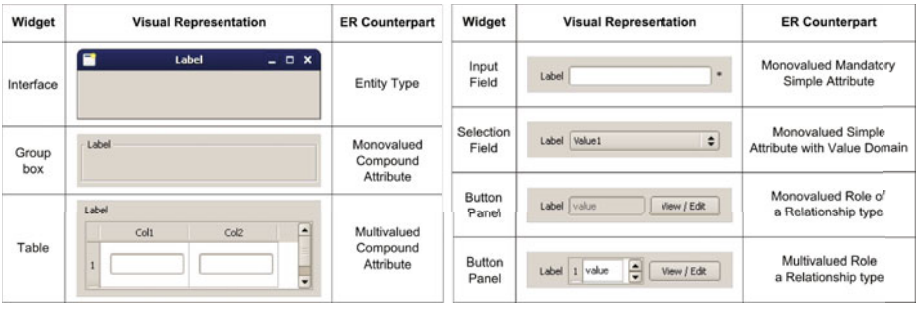

Fig. 2. Available graphical widgets with their mapping rules to data structures

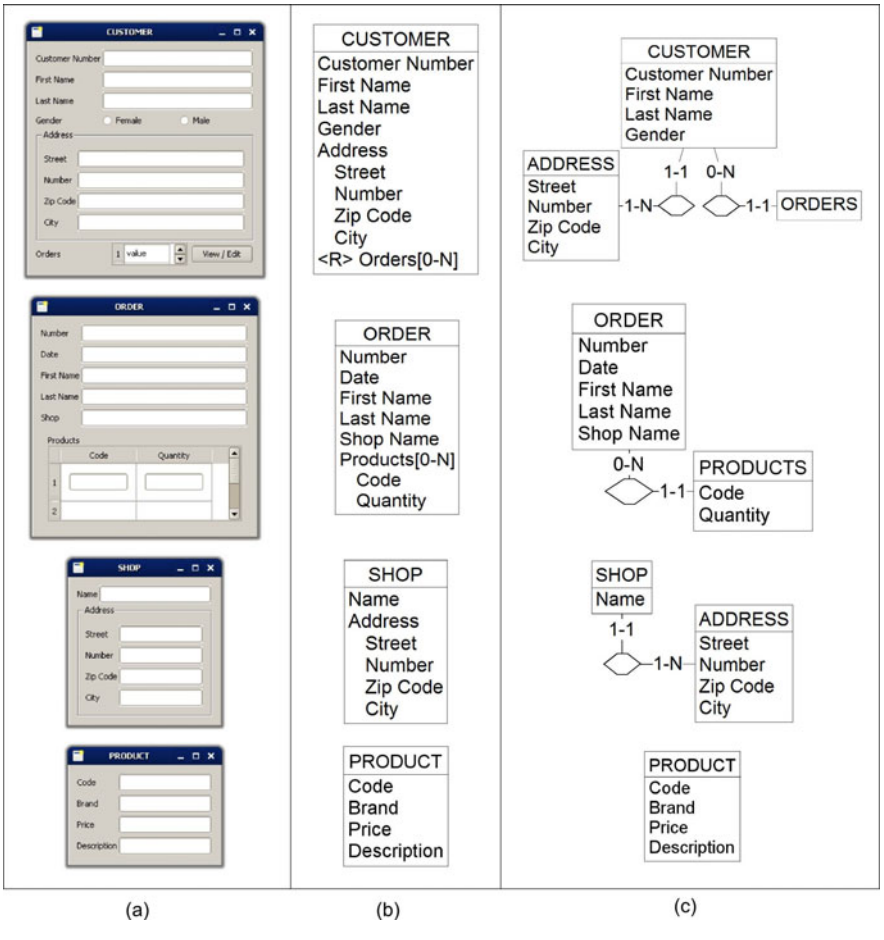

Fig. 3. (a) Example of user-drawn interfaces. (b) Translation of the interfaces into raw entity types. (c) Translation of the raw entity types into independent schemas.

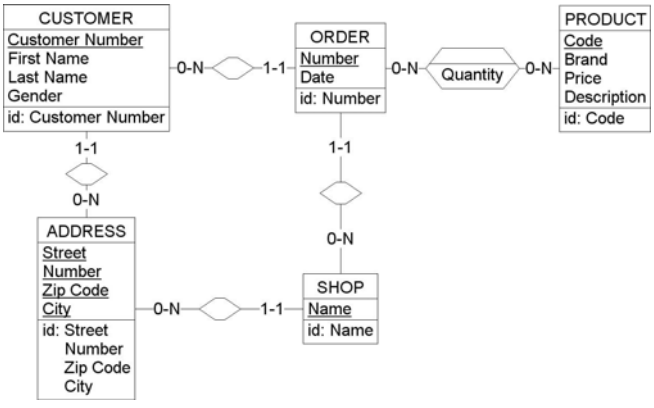

Fig. 4. Integrated schema of our running example

3. Investigate: Cross-analyzing each individual schema usually brings to light possible ambiguities as well as redundant information in the interfaces. In particular, semantic and syntaxic redundancies are automatically identified and presented to end-users for manual validation.

4. Nurture: To elicit possible constraints and dependencies among data structures, induction techniques are applied on positive and negative data samples provided by end-users.

5. Bind: Validated redundancies, constraints and dependencies are processed to integrate the individual schemas into a pure conceptual schema that represents the data requirements. A dedicated approach has been developed, based on transformational techniques. Fig. 4 illustrates the result of the integration for our running example.

6. Objectify: A lightweight prototype application is generated from the integrated conceptual schema. It comprises a simple data manager that uses the interfaces drawn by the end-users and allows them to manipulate the concepts that have been expressed, typically to inspect, create, modify and remove data.

7. Wander: Finally, the end-users are invited to *"play" with the prototype* in order to ultimately validate the requirements, or identify remaining flaws.

5 Methodological Specificities

In this section, we describe in more detail four basic techniques at the core of the RAINBOW approach, namely Reverse engineering, Modular refinement, View integration and Transformational approach. We will show how the context in which they are used has lead to specialize them.

5.1 Reverse Engineering

Reverse engineering consists, among other things, in recovering or reconstructing the functional specifications from a piece of software, starting mainly from the

source code of the programs [18,19]. Such a process is typically required when an existing database has to be refactored or migrated toward a different technology. Reverse engineering hence aims at recovering a conceptual schema that is the most faithful to the original one, working from multiple system artifacts, such as documentation (when available), the DDL code of the database, data instances, screens, reports and forms, source code of application programs.

However, our objective is here to "build the truth" rather than "find the truth", as in traditional Reverse engineering situations. In particular, the interfaces are used as a specification language as opposed to the usual Reverse engineering of existing screens. This requires to significantly adapt the usual database Reverse engineering (DBRE) methodology [20]. Indeed, as depicted in Figure 5 (a), DBRE typically comprises the following four sub-processes: (1) *Physical extraction*, which consists in parsing the DDL code in order to extract the raw physical schema of the database; (2) *Refinement*, which enriches the raw physical schema with additional constructs and constraints elicited through the analysis of the application programs and other sources; (3) *Cleaning*, which removes the physical constructs (such as indexes) for producing the logical schema; (4) *Conceptualization*, which aims at deriving the conceptual schema that the logical schema implements.

Such a methodology is not applicable as is in the context of the RAINBOW approach, as shown in Figure 5 (b). Starting from a set of user interfaces

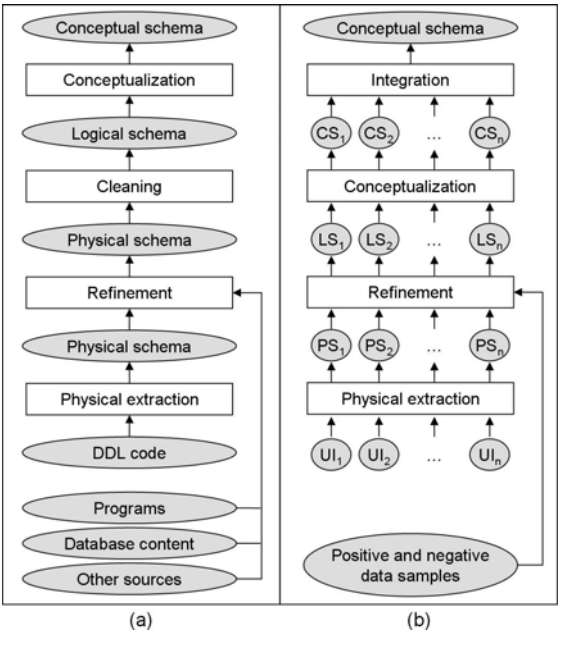

Fig. 5. (a) Standard database Reverse engineering methodology. (b) Reverse engineering methodology of the RAINBOW approach.

$(UI_1, UI_2, \cdots, UI_N)$, the physical extraction does not allow one to derive a complete physical schema, but a *set* of *partial views* of this schema $(PS_1, PS_2, \cdots, PS_N)$. Similarly, the refinement process may not rely on additional available artifacts such as application programs or database contents. However, it can take benefit from data samples provided by the users through the interfaces they have drawn, leading to the identification, among others, of candidate dependency constraints and attribute domains. The recovered constraints, once validated, are used to enrich the physical schemas PS_i in order to obtain a set of logical schemas LS_i. The cleaning phase, as defined above, does not make sense in the absence of an initial DDL code. Instead, the conceptualization step allows one to derive a set of partial conceptual schemas (CS_i) from the logical schemas obtained so far. In particular, the logical schemas are *normalized* in order to ease the identification of similarities between them. This important process relies on transformation techniques that will be developed in Section 5.4. During the integration phase, the partial conceptual schemas are merged, based on structural and semantic similarity criteria. This process, further described in Section 5.3, produces a single complete conceptual schema.

5.2 Modular Refinement

One of the key assets of our approach is its flexibility, especially regarding the enrichment of the data models. As we have seen, proficient end-users can already provide technical properties (such as size and type of attributes, domain of values, existence constraints, reference keys, ...) during the drawing phase. For non expert end-users, such properties can be discovered later on, as well as functional dependencies, from a set of positive and negative data samples [21,22] provided by end-users. Several algorithms have been proposed to support such a task [23]. However, a major limitation of these techniques comes from the fact that they rely on massive preexisting data sets, whereas we are working in a context where there is potentially no pre-existing data (or the (re)encoding cost would be too high). We therefore use a simplified version of such a mining algorithm to build Armstrong relations [24], by first asking end-users to provide data samples through the interfaces they have drawn. From these data, candidate constraints and dependencies are identified. Subsequently, using additional data samples and generated examples that the end-users may validate or reject, the constraints and dependencies are progressively enforced, discarded or refined for each entity type.

5.3 View Integration

In this section, we focus on the non standard integration process of our method. First of all, elements of integration (ambiguities, similarities, ...) are gradually collected, in order to be resolve later on.

The first element that retains our attention is the *semantic redundancy and ambiguity*. This issue arises due to the previously mentioned limitations of written natural language and possible mistakes, which lead to unclear labels in the

interfaces. In our example, one can for instance notice the closeness of the labels `Orders` (from the interface `Customer`) and `Order`. Moreover, consider for instance the strings "primary provider" and "alternative supplier", which could be used as labels. They are not lexically close, but one may notice the nearness of meaning of the words "provider" and "supplier".

Identifying similar labels and words is a well-known problem that can be dealt with using *String Metrics* [25], ontologies and similarity dictionaries. First of all, we extract the set W_i of relevant words contained in each label l_i of the interfaces elements, by typically casting away articles and conjunctions. Then, by combining these techniques, we identify ambiguously labeled elements within user-drawn interfaces.

For a given string distance metric s and threshold t, we declare two labels l_1 and l_2 as *lexically similar* iff :

$$s(l_1, l_2) \leq t \ \vee \ \exists \, w_{1_i} \in W_1, w_{2_i} \in W_2 : s(w_{1_i}, w_{2_i}) \leq t$$

Within a set of labels $L = \{l_1, l_2, ..., l_n\}$, we also define a *subset of lexically similar labels* L_i as a subset $\{l_{i_1}, l_{i_2}, ..., l_{i_m}\} \subseteq L$ verifying:

$$\forall \, l_{i_j} \in L_i, \exists \, l_{i_k} \in L_i : \text{“}l_{i_j} \text{ and } l_{i_k} \text{ are } \textit{lexically similar}\text{”}$$
$$\wedge \, \forall \, l_{i_j} \in L_i, l_k \in L \backslash L_i : \text{“}l_{i_j} \text{ and } l_k \text{ are not } \textit{lexically similar}\text{”}$$

Among the wide variety of reliable string distance metrics, Jaro-Winkler's distance d_w [26] has proved to be a good fit for short strings. It uses a prefix scale which gives more favorable ratings to strings that match from the beginning for a set prefix length. We use a variant s_w of this distance, also taking the longest suffix into account. We define it for two strings s_1 and s_2 as:

$$s_w(s_1, s_2) = 1 - max(d_w(s_1, s_2), d_w(s_1^{'}, s_2^{'}))$$
$$\text{where } s_i^{'} \text{ is the reversed version of the string } s_i$$
$$\text{and } t_w = 0.2 \text{ was found to be a reasonable threshold.}$$

Since we want to partition the set L of all the labels available in the user-drawn interfaces into subsets of semantically similar labels, we confront each label to the others using s_w for string distance and the lexical reference system WordNet [27] for synonymy. We hence build the set of the lexically similar subsets, then visually point out the discovered similarities between concepts in the user-drawn interfaces in order to ask the end-users to validate or reject them.

Another issue concerns *structural redundancy*, which occurs when two entity types share a *pattern*. We define a pattern as a bijection between two sets of attributes belonging to different entity types. For instance, `Customer` and `Order` share a pattern based on the labels `First Name` and `Last Name`. The similarity between pairs of attributes from each set is measured using several indicators (e.g., the label). For each indicator, we define a *similarity index*, the values of which fall between 0 (strictly different) and 1 (strictly identical). The similarity of two attributes is computed as a weighted average of the similarity indicators.

Given the hierarchical structure of the interfaces, and thus the tree-like structure of the underlying models, the problem of extracting structural redundancies constitutes a particular case of *frequent embedded subtrees mining in rooted unordered trees*, which is similar to pattern mining in XML documents. This complex issue is described by Jimenez *et al.* [28], who also list related algorithms such as Zaki's SLEUTH or Asai *et al.*'s UNOT. In this context, entity types can be seen as root nodes, compound attributes as intermediary nodes, simple attributes as leaves, and the attribute order is irrelevant, as we explored in [5].

While tree-based approaches are suitable for complex and deep graphs, we observe that the structure of user-drawn interfaces is usually quite simple (no more than four levels of imbrication), if only by concerns of legibility and usability [10]. Instead of considering such heavy algorithms, we use a simple algorithm that consists in comparing one by one each entity type to elicit patterns and visually point them out. The end-users are then invited to arbitrate them by classifying the relation between the concepts sharing a pattern among one of these most usual cases: equality (the entity types represent the same concept), union (the entity types partially represent the same concept, which may translate the specialization of a higher-level concept non explicitly expressed), comprehension (one of the entity types is a specialization of the other), complementarity (one of the entity types actually refers to the other) or difference (the entity types fortuitously share a set of attributes).

Once the knowledge of the domain is enriched thanks to the end-user input, that validates or rejects ambiguities, similarities and dependencies, the integration process can take place. Transformational techniques have proved to be particularly powerful to carry out this task, which is a typical case of *database schema integration* [29]. They enable the integration of similar objects into a unique, non-redundant structure, without any loss of semantics. For simplicity, we consider in this paper that two similar objects refer to the same concept and can therefore be merged. Hence, the entity types of the logical schemas are integrated pairwise, and whenever needed, end-users are invited to choose which attributes of the two objects are relevant and should thus be kept during the merging process. However, as mentioned in the structural analysis, there is not always a strict identity between two concepts and other integration techniques must then be used to resolve integration conflicts [30]. At this point, it is clear that this process cannot be fully automated and that the analysts must be actively involved.

5.4 Transformational Approach

Finally, our approach heavily relies on the *transformational engineering* paradigm, according to which most (if not all) database engineering processes can be modeled as a chain of schema transformations [7]. A transformation operator is defined by a rewrite rule that substitutes a target schema construct for a source construct. The most interesting operators are said *semantics-preserving*, in that the source and target constructs convey the same semantics. Recall for instance Fig. 1. The schema on top of subfigure (a) is claimed to be a more

expressive but equivalent version of the schema on the right of subfigure (b), the latter representing the information contents of the electronic form on the left of subfigure (b). Two questions naturally arise: (1) how has the schema with entity types `Department`, `Employee`, `Clerk` and `Worker` been produced from the sole `Employee` aggregate and (2) what guarantee do we have that both structures are equivalent?

Fig. 6 illustrates two important operators in the context of user interface Reverse engineering, namely *attribute to entity type mutation* and *upward inheritance*. Both are *reversible* or *semantics preserving*, so that they can be applied from left to right and from right to left. The first one (T1) transforms an entity type into an equivalent attribute (and conversely). The second transformation (T2) integrates the subtypes of an entity type as complex attributes of the latter (and conversely). They lack some necessary pre- and post-conditions to be fully semantics preserving, but they are sufficient considering the scope of this paper.

When applied successively to the schema of Fig. 1(a), transformation T1 (on entity type `Department`) and transformation T2 (on subtypes `Clerk` and `Worker`) yield the schema of Fig. 1(b). Since the transformations are semantics-preserving, their inverse can be applied to the schema of Fig. 1(b), which is transformed in that of Fig. 1(a). This simple scenario illustrates the use of transformations in the Conceptualization process. It also shows how the expressive schemas of Fig. 3(c) can be extracted from raw physical schemas of Fig. 3(b) by a chain of semantics-preserving transformations.

Besides, ER schemas, be they conceptual or logical, can be given a formal semantics in several ways. Fig. 1(a) suggests an approach based on an extended NF^2 relational model. This semantics makes it easy to demonstrate the equivalence of two schemas by building a chain of algebraic operators, such as project/join and nest/unnest [7]. This formal framework is essential to evaluate the applicability of transformation in specific contexts, notably to identify missing parts in pre-conditions. For example, the functional dependency `DeptNbr` → `Location` in Fig. 1(b) is a part of the pre-condition to recover the `Department` entity type. Should this property be missing, the resulting schema would have been different (as suggested by Fig. 6(a)).

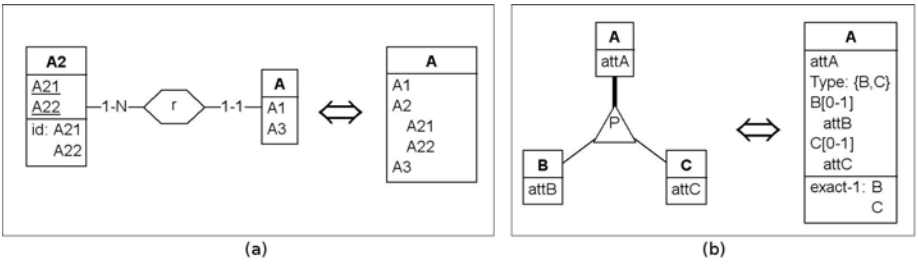

Fig. 6. T1 (a) and T2 (b), variants of two (almost) semantics-preserving transformations

6 Tool Support

The RAINBOW Tool Kit is a user-oriented development environment, intended to assist end-users and analysts in the definition and validation of database requirements through prototyping. It therefore supports the first steps of our approach by offering ready-to-use widgets and mapping rules with the ER model. The tool kit interacts with the repository of DB-Main, a database engineering CASE Tool [31] providing all the necessary functionalities to support a complete database design process (from conceptual analysis to DDL code generation). It provides transformation tools and supports database Reverse engineering. The interaction between these tools allows one to cover the whole database engineering process from both the end-user and the analyst perspectives.

7 Concluding Remarks

7.1 Contributions

This paper has presented a comprehensive interactive approach to bridge the gap between end-users and analysts during the requirements analysis phase of database engineering. This approach supports the elicitation and validation of static data requirements with end-users, while overcoming several limitations of existing prototyping methods. It relies on the expressiveness and understandability of form-based user interfaces, used jointly with tailored Reverse engineering techniques to acquire data specifications from existing artifacts. We offer a very simple interface model, inspired by high level abstract models such as UsiXML [32]. Since any interface widget dedicated to data representation can be expressed using these basic components, limiting the number of available widgets appears to ease the user interaction without restricting it. The process of drawing and specifying the interfaces takes in account possible labeling variations, and offers an incremental and flexible enrichment of the underlying data models.

The RAINBOW approach lies within broader perspectives, such as the ReQuest Framework. While offering a precious user empowerment and involvement in the data requirements elicitation, it provides an expressive and interactive part of user requirements thanks to the RAINBOW Tool Kit and its connection to DB-Main. The requirements are materialized as a documented application domain (the conceptual model) and a documented database (DDL, queries, etc.). The simple yet operational generated prototype can serve as a basis for further application development over the database.

7.2 Limitations and Future Work

Although our approach addresses a significant subset of data requirements, it does not cover all of its aspects, typically the dynamic ones. Therefore, our approach does not replace more traditional task and information analysis approaches, but rather complements them. For instance, the form-based graphical representation of the underlying data model can be used during interviews to

stimulate the discussion. As for the generated prototype, it can be used during the task analysis to capture real-time use cases and define the expected behavior of the system. In addition, analyzing how the tasks are performed using the prototype in comparison to the legacy information system (if any), can help to support the Reverse engineering of existing artifacts and even induce more general considerations on the definition of the target information system.

This work assumes that end-users are able, with proper training, to represent simple and intuitive concepts. However, while limiting the approach to basic data structures can be seen as a "positive" simplification, we intend to push our investigation further to allow the users to also express complex structures such as temporal or semi-structured data.

A more general issue concerns the feasibility of involving different levels of users in our approach. A software engineering process may indeed involve stakeholders ranging from casual (or even novices) to experts users, such as analysts, designers or programmers, all of which may lack proper expertise in Database engineering. Still, to ensure the quality of the requirements, none of these users can be cast away, and each of them should be provided with adequate means to express their needs. We therefore intend to continue working on our approach and our tools, in order to make them even more intuitive and adaptative, according to the category of all our potential users.

Acknowledgments. This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship. Partial support was also received from the Région Wallonne (through the ReQuest project) and the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy (through the MoVES project).

References

1. Rosson, M.B., Carroll, J.M.: Usability Engineering: Scenario-Based Development of Human-Computer Interaction. Morgan Kaufmann, San Francisco (October 2001)
2. Fischer, G.: Beyond 'couch potatoes': From consumers to designers and active contributors. First Monday 7 (2002)
3. Vosburgh, J., Curtis, B., Wolverton, R., Albert, B., Malec, H., Hoben, S., Liu, Y.: Productivity factors and programming environments. In: ICSE, pp. 143–152 (1984)
4. Brogneaux, A.F., Ramdoyal, R., Vilz, J., Hainaut, J.L.: Deriving user-requirements from human-computer interfaces. In: Proc. of 23rd IASTED Int'l Conf., pp. 77–82 (2005)
5. Vilz, J., Brogneaux, A.F., Ramdoyal, R., Englebert, V., Hainaut, J.L.: Data conceptualisation for web-based data-centred application design. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 205–219. Springer, Heidelberg (2006)
6. Batini, C., Ceri, S., Navathe, S.B.: Conceptual database design: an Entity-relationship approach. Benjamin-Cummings Publishing Co., Inc. (1992)
7. 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)

8. Pizano, A., Shirota, Y., Iizawa, A.: Automatic generation of graphical user interfaces for interactive database applications. In: Proc. of the 2nd Int'l Conf. on Information and Knowledge Management (CIKM 1993), pp. 344–355. ACM, New York (1993)
9. Shoval, P., Shiran, S.: Entity-relationship and object-oriented data modeling-an experimental comparison of design quality. Data Knowl. Eng. 21(3), 297–315 (1997)
10. Choobineh, J., Mannino, M.V., Tseng, V.P.: A form-based approach for database analysis and design. Communications of the ACM 35(2), 108–120 (1992)
11. Terwilliger, J.F., Delcambre, L.M.L., Logan, J.: The user interface is the conceptual model. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 424–436. Springer, Heidelberg (2006)
12. Rollinson, S.R., Roberts, S.A.: Formalizing the informational content of database user interfaces. In: Ling, T.-W., Ram, S., Li Lee, M. (eds.) ER 1998. LNCS, vol. 1507, pp. 65–77. Springer, Heidelberg (1998)
13. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: Proc. of the Conf. on The Future of Software Engineering, pp. 35–46. ACM Press, New York (2000)
14. Davis, A.M.: Operational prototyping: A new development approach. IEEE Softw. 9(5), 70–78 (1992)
15. Kösters, G., Six, H.W., Voss, J.: Combined analysis of user interface and domain requirements. In: Proc. of the 2nd Int'l Conf. on Requirements Engineering, pp. 199–207. IEEE Computer Society, Los Alamitos (1996)
16. Yang, F., Gupta, N., Botev, C., Churchill, E.F., Levchenko, G., Shanmugasundaram, J.: WYSIWYG development of data driven web applications. Proc. of the VLDB Endowment 1(1), 163–175 (2008)
17. Ravid, A., Berry, D.M.: A method for extracting and stating software requirements that a user interface prototype contains. Requir. Eng. 5(4), 225–241 (2000)
18. Chikofsky, E.J., Cross, J.H.: Reverse engineering and design recovery: A taxonomy. IEEE Software 7(1), 13–17 (1990)
19. Hall, P.A.V.: Software Reuse and Reverse Engineering in Practice. Chapman & Hall, Ltd., Boca Raton (1992)
20. Hainaut, J.L.: Introduction to database reverse engineering. LIBD Publish (2002), `http://www.info.fundp.ac.be/~dbm/publication/2002/DBRE-2002.pdf`
21. Tseng, V.P., Mannino, M.V.: Inferring database requirements from examples in forms. In: Proc. of the 7th Int'l Conf. on ER Approach, pp. 391–405 (1988)
22. Ram, S.: Deriving functional dependencies from the entity-relationship model. Commun. ACM 38(9), 95–107 (1995)
23. Yao, H., Hamilton, H.J.: Mining functional dependencies from data. Data Min. Knowl. Discov. 16(2), 197–219 (2008)
24. De Marchi, F., Petit, J.M.: Semantic sampling of existing databases through informative armstrong databases. Information Systems 32(3), 446–457 (2007)
25. Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: Proc. of IJCAI 2003 Workshop on Information Integration on the Web (IIWeb 2003), pp. 73–78 (2003)
26. Winkler, W.E.: String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In: Proc. of the Section on Survey Research Methods, pp. 472–477 (1990)

27. Fellbaum, C.: WordNet: An Electronic Lexical Database. MIT Press, Cambridge (1998)
28. Jiménez, A., Berzal, F., Cubero, J.C.: Mining induced and embedded subtrees in ordered, unordered, and partially-ordered trees. In: An, A., Matwin, S., Raś, Z.W., Ślęzak, D. (eds.) ISMIS 2008. LNCS (LNAI), vol. 4994, pp. 111–120. Springer, Heidelberg (2008)
29. Batini, C., Lenzerini, M., Navathe, S.B.: A comparative analysis of methodologies for database schema integration. ACM Computing Surveys 18(4), 323–364 (1986)
30. Spaccapietra, S., Parent, C., Dupont, Y.: Model independent assertions for integration of heterogeneous schemas. The VLDB Journal 1(1), 81–126 (1992)
31. DB-Main: The official website, `http://www.db-main.be`
32. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V.: Usixml: A language supporting multi-path development of user interfaces. In: EHCI/DS-VIS, pp. 200–220 (2004)