Interactively Eliciting Database Constraints and Dependencies

Ravi Ramdoyal and Jean-Luc Hainaut

Laboratory of Database Application Engineering - PReCISE Research Centre
Faculty of Computer Science, University of Namur
Rue Grandgagnage 21 - B-5000 Namur, Belgium
{rra,jlh}@info.fundp.ac.be
http://www.fundp.ac.be/precise

Abstract. When designing the conceptual schema of a future information system, it is crucial to define a set of constraints that will guarantee the consistency of the subsequent database once it is implemented and operational. Eliciting and expressing such constraints and dependencies is far from trivial, especially when end-users are involved and when there is no directly usable data to play with. In this paper, we present an interactive process aimed to elicit hidden constraints such as value domains, functional dependencies, attribute and role optionality and existence constraints. Inspired by the principles of Armstrong relations, it attempts to acquire minimal data samples in order to validate declared constraints, to elicit hidden constraints and to reject irrelevant constraints in conceptual schemas. This process is part of the RAINBOW approach, destined to develop the data model of an information system based, among others, on the reverse engineering of user-drawn form-based interfaces.

Keywords: Information Systems Engineering, Requirements Engineering, Database Engineering, Electronic Forms Reverse Engineering, Constraint Discovery.

1 Introduction

In the realm of Requirements engineering, Database engineering focuses on data modelling, where the static data requirements are typically expressed by means of a conceptual schema, which is an abstract view of the static objects of the application domain. There are numerous types of constraints and dependencies that can be established for such a schema. They can concern individual elements, their components, or even how (the components of) an element can affect (the components of) other elements. Traditional database elicitation techniques, such as the analysis of corporate documents and interviews of stakeholders, usually yield many relevant constraints during the design of the conceptual schema, however some constraints may be forgotten, typically because the domain experts were not aware of them, or (more probably) because they are part of some tacit knowledge.

Though the necessity to associate end-users of the future system with its specification and development steps has long been advocated [1], several approaches rather propose to deal with the discovery of such constraints by analysing the content of a related database (or at least, a set of relevant data samples). However, they rely on the preexistence of large sets of data, which can obviously be problematic in the process of designing a new information system: there might be no usable legacy database, or gathering and reencoding a significant amount of data would be unrealistic.

In this context, the RAINBOW approach [2] provides an alternative and interactive process based on the analysis of a limited set of user-provided data samples in order to elicit and suggest database constraints and dependencies for a given schema. RAINBOW is a collaborative and interactive user-oriented approach to develop the static data model of an information system based on the reverse engineering of user-drawn form-based interfaces. It relies on the adaptation and integration of principles and techniques coming from various fields of study, ranging from Database Forward and Reverse Engineering to Prototyping and Participatory Design.

In this paper, we present how to use the RAINBOW approach to discover constraints and dependencies. In particular, we focus on the elicitation of functional dependencies, which are a fundamental and critical aspect of conceptual modelling that has proved difficult to apprehend. 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. Section 5 discusses the evaluation of this process, while Section 6 discusses the proposal and concludes this paper.

2 Research Context

2.1 The RAINBOW Approach

The RAINBOW approach is a collaborative and interactive user-oriented approach to design database conceptual schemas in the context of Information System engineering [2]. It exploits the expressiveness of user-drawn form-based interfaces and prototypes, and specialises and integrates standard techniques to help acquire and validate data specifications from existing artefacts in order to use such interfaces as a two-way channel to communicate static data requirements between end-users and analysts. The approach is formalised by a semi-automatic seven-step process dealing with the progressive modelling of the application domain:

1. *Represent*: the end-users are invited to draw and specify a set of form-based interfaces to perform usual tasks of their application domain;
2. *Adapt*: the forms are "translated" into data models, which basically consists in extracting a data model from each interface using mapping rules;
3. *Investigate*: the data models are cross-analysed to highlight and arbitrate semantic and structural similarities and produce a pre-integrated schema;

4. *Nurture*: using the interfaces that they drew, the end-users are invited to provide data examples that are analysed to infer and arbitrate possible constraints and dependencies;
5. *Bind*: the pre-integrated schema is completed and refined into a non redundant integrated conceptual schema;
6. *Objectify*: from the integrated conceptual schema, the artefacts of a prototypical data manager application are generated;
7. *Wander*: finally, the end-users are invited to play with the prototype in order to refine and ultimately validate the integrated conceptual schema.

In order to position end-users as major stakeholders throughout the data requirements process, the approach uses form-based interfaces as a controlled basis for joint development, analysis and discussion. In particular, in order to make the development of the interfaces more accessible and focus the drawing on the substance rather than (ironically) the form, the available graphical elements are restricted to the most commonly used ones (`forms`, `fieldsets` and `tables`, `inputs`, `selections` and `buttons`) and limited the layout of forms as a vertical sequence of elements, which also simplifies the transition from the form model and the ER model. This drawing phase is supported by the RAINBOW Toolkit, which is the dedicated and integrated tool support intended to assist end-users and analysts during the different RAINBOW processes.

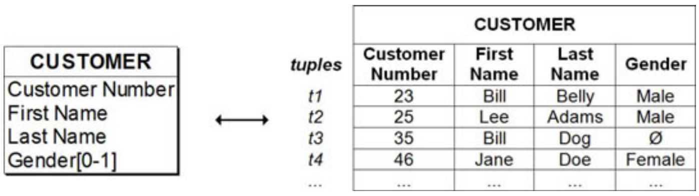
The interfaces being drawn by non experts and possibly multiple end-users increases the possible inconsistencies among the individual labels and the structures used in the forms [2]. The semantic and structural similarities are therefore analysed to manage commonality and standardise the form constructs and their underlying data counterpart. Semantic similarities arise due to the richness of written natural language, which can lead to spelling and meaning ambiguities. Structural similarities which occurs when two entity types share a pattern, which is a bijection between two sets of attributes belonging to different entity types. RAINBOW deals with the elicitation and subsequent unification of semantic similarities using String Metrics, Ontologies and dictionaries. Though the forms and their underlying data models have a tree-like arborescence, their structure is simple and does not necessitate using complex techniques such as tree mining approaches to discover structural similarities. The shared patterns are instead elicited by comparing the different entity types, and the structures are subsequently unified according to the meaning of the pattern (equality, union, comprehension, complementarity, composition or difference).

The RAINBOW approach also deals with the integration of each schema corresponding to a form into a single normalised schema representing the domain of application, before generating and testing the resulting associated applicative components. However, before leading these processes, an important step consists in eliciting the relevant constraints and dependencies of the domain of application.

2.2 Constraints and Dependencies in Conceptual Schemas

When designing a conceptual schema, it is indeed important to define a set of constraints that will guarantee that once the subsequent database is implemented and operational, any change made to its content by authorised users will maintain its consistency. Typically, inserting, modifying or deleting values from the database should not result into data anomalies or unnecessary redundancies. For this purpose, let us introduce the *relational model* of a database according to the *First normal form* (1NF), which is a database model based on first-order predicate logic, first formulated and proposed by Codd [3]. In the relational model, all the data is represented through *relation*s (also known as *tables*). A relation is composed of *attribute*s (a.k.a. as *columns*), each of which is defined on a *domain*, which is a given set of values. A *tuple* (a.k.a. *row*) contains all the data of a single instance, that is, a value for each attribute of the relation. Relations and attributes of the relational model will be used to model entity types and attributes in the GER model[1], as illustrated in Figure 1.

The transition between conceptual and relational models is facilitated in the RAINBOW approach since the complex conceptual schemas obtained from the form-based interfaces are recursively transformed until they reduce to simple schemas including flat entity types (with elementary attributes) and binary relationship types. Coping with these relationship types in the 1NF relational modeling requires a little trick; the roles are represented through *role attributes*, so that they appear as attributes for which the value belongs to the possible tuples of the entity type associated through the binary relationship type. In addition, when no value is provided in a form for a given attribute, a default "empty" value (noted ∅) is encoded, so that null values are avoided.



tuples	Customer Number	First Name	Last Name	Gender
t1	23	Bill	Belly	Male
t2	25	Lee	Adams	Male
t3	35	Bill	Dog	Ø
t4	46	Jane	Doe	Female
...	...	...	...	...

Fig. 1. The representation of a Customer using the GER and relational model

Among the many *constraints* usually found in database schema, we have selected the following ones. *Domains of values* restrict the possible values of given attributes, for instance using domain types, sets or ranges of (un)authorised values, rule-based formulas for the values, and so on. For instance, a `Customer Number` may be restricted to integer values, and the `Gender` limited to the set of values $\{Female, Male\}$. *Cardinality constraints* define the minimal (typically

[1] The Generic Entity-Relationship (GER) model is a wide-spectrum model used to describe database schemas in various abstraction levels and paradigms [4].

zero or one) and maximal numbers (typically one or infinite) of occurrence(s) of given attributes and roles. *Existence constraints* define how optional components (attributes and roles) may influence each other. For two components A and B, these constraints can be:

- *coexistence*, which implies that A and B must always be not null simultaneously;
- *exclusion*, which implies that A and B cannot be not null simultaneously;
- *at-least-one*, which implies that A and B cannot be null simultaneously;
- *exactly-one*, which implies that if A is not null, then B should be null, and vice-versa;
- *implication*, where A implies B means that A can be not null only if B is not null itself;

For a relation, an *identifier* (a.k.a. *candidate key*) is a set of attributes so that, when considering all the possible tuples of the relation, there cannot be more than one tuple having the same combination of domain values for these attributes. For instance, from the tuples visible in Figure 1, we could assume that `Customer Number` forms an identifier for relation `Customer`, since there are no two tuples with the same value of this attribute.

A similar notion is the concept of *functional dependencies*, which are materialised by the explicit or implicit constraints between two sets of attributes in a relation. Given relation R, a set of attributes $X \subseteq R$ is said to functionally determine another set of attributes $Y \subseteq R$, if and only if all the tuples with the same combined values of X also have the same combined values of Y. This functional dependency is written $R : X \rightarrow Y$, with X and Y respectively called the *left-hand side* (a.k.a. *determinant*) and *right-hand side* (a.k.a. *dependent*) of the functional dependency f. For instance, from the tuples visible in Figure 1, it seems that the functional dependency `Customer:First Name` → `Last Name` does not hold, since there are two persons named “Bill” but with a different family name. On the other hand, the functional dependency `Customer:First Name, Last Name` → `Gender` could hold, but would need to be validated.

3 State of the Art in Constraints and Functional Dependencies Mining

Analysing the content of a database or a subset of data samples can intuitively lead to make plausible assumptions on, e.g., the domains of values, the cardinalities of the attributes, their existence constraints and possibly their identifiers. Let $t[\mathcal{C}]$ be the restriction of a tuple t to the set of components $\mathcal{C}$ (called *projection* of t onto $\mathcal{C}$), and $t[C]$ be the restriction of t onto a given component C. Now consider for instance an optional textual attribute A: if for any tuple t_i, we observe that $t_i[A]$ is never null and always composed of a number, we could easily wonder if A is not actually a mandatory numeric attribute. Moreover, if all the t_i have different values A, this could suggest that A is in fact an identifier. The same kind of *induction* can be applied on optional attributes to assess their possible existence constraints. However, functional dependencies mining is far less trivial.

Back in 1995, Ram presented four types of approaches to derive functional dependencies from an existing conceptual ER schema [5]. The first category consists in using keyword analysis to identify intra-entity functional dependencies: typically, attributes bearing a suffix or prefix such as "id" or "number" should be considered potential determinants, while attributes bearing a suffix or prefix such as "maximum", "minimum", "average" or "total" should be considered potential dependent attributes. The second category consists in analysing the cardinalities of binary relationships to identity inter-entity functional dependencies, typically between their identifiers. The third category is similar, but concerns N-ary relationships. And finally, the fourth category consists in analysing sample data to elicit undiscovered functional dependencies. These principles were supported by the FDExpert tool.

The first three categories rely on the analysis of the schema itself, while the latter category, known as the *dependency discovery problem*, focuses on the content of the database. The latter category is a standard issue, especially in data mining, database archiving, data warehouses and Online Analytical Processing (OLAP). The most prominent existing algorithms dealing with this issue can be classified in three categories, that are difficult to compare qualitatively [6,7].

The first two categories basically try to explore the search space (i.e. the possible combinations of the attributes for a given relation) in the most efficient way possible, in order to test the associated functional dependencies using a stripped partition database computed from that relation. The *candidate generate-and-test* approach progressively explores and prunes the search space in a levelwise manner, while partitioning the database using attribute-based equivalence classes, as in Huhtala et al.'s TANE [8], Novelli and Cicchetti's FUN [9], or Yao and Hamilton's FD_Mine [7]. The *minimal cover* approach structures the search space using hypergraphs that are explored to discover the minimal cover of the set of FDs for a given database, i.e. the minimal set of FDs from which the entire set of FDs can be generated using the Armstrong axioms, as in DepMiner, proposed by Lopes et al. [10] and FastFDs, proposed by Wyss et al. [11].

Finally, *Formal concept analysis* (FCA) has also been used recently to find and represent logical implications in datasets [12], mainly through a closure operator from which concepts (closed sets) can be derived. For instance, Baixeries uses Galois connections and concept lattices as a framework to find functional dependencies [13], while Rancz et al. optimise an existing method introduced by [14], which provides a direct translation from relational databases into the language of power context families, in order to build inverted index files to optimise the elicitation the functional dependencies in a relational table through the construction of their formal context [15]. The latter authors also developed the subsequent FCAFuncDepMine software to detect functional dependencies in relational database tables [16]. Similar principles were also used in Flory's method, which was based on the definition and analysis of a *matrix* and its associated *graph* of functional dependencies [17].

4 An Interactive Process to Elicit Constraints and Functional Dependencies

4.1 Overview

As we have seen, traditional elicitation techniques may neglect constraints, while dependency discovery approaches rely on massive pre-existing data sets, which is problematic when there is no data samples available, or when their re-encoding would be too expensive. To tackle this problem, the RAINBOW approach proposes to use form-based interfaces that were previously drawn by the end-users themselves in order to let them provide a limited set of data samples from which constraints and dependencies could be inferred and suggested. Though such constraints can be provided directly, it appears that the interactive acquisition and processing of data samples is useful and more natural in this process, as it also helps to visualise the implications of existing constraints.

In this section, we present an interactive process inspired by the principles of *Armstrong relations*, which are relations that satisfy each functional dependency implied by a given set of functional dependencies, but no functional dependency that is not implied by that set [18]. This twofold process focuses on eliciting the constraints and dependencies mentioned in Section 2.2, i.e. domains of value, cardinalities, existence constraints, identifiers and functional dependencies. On the one hand, data samples are acquired to restrict the potentially "hidden" constraints, and on the other hand, potential constraints are arbitrated to conversely restrict the tuples that can be encoded. Some of these properties can be trivial and may be expressed directly, or have been expressed during the preliminary drawing of the form-based interfaces. For instance, in the form of Fig. 2, the `Last Name` of a `Person` is mandatory, while its `Title` appears to be optional. Likewise, the `Birth date` has been encoded as a date value, while the `Zip code` of the `Contact` may have been encoded as a textual value. However, the specified properties may need to be refined (for instance, the `Zip code` may prove to be numeric), and there may be some unsuspected constraints and dependencies among the elements of the schema.

We therefore propose to start by envisaging initial potential constraints and dependencies. Then, using user input, we progressively *validate* or *discard* them, and generate alternatives until they are all arbitrated. This process hence relies on several sub processes:

– the initialisation of all the currently *declared* (explicitly expressed during the drawing step) and *potential* (implicitly verified by the present set of tuples) constraints and dependencies;
– the acquisition and analysis of new valid data samples in order to automatically discard the invalid potential constraints and dependencies, and possibly generate acceptable potential alternatives;
– the arbitration of potential constraints and dependencies through user *validation* or *discardure*, and the subsequent generation of new potential alternatives;

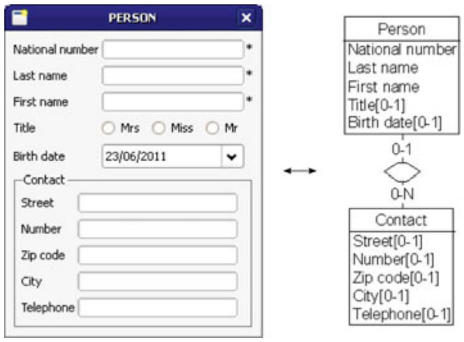


Fig. 2. A form-based interface describing a "Person" and its associated conceptual schema, using the GER representation[2]

– the processing of the validated constraints and dependencies, once there are no other potential constraints or dependencies left.

The acquisition of data samples progressively restricts the set of potential constraints, while conversely, validating constraints also restricts the future data samples that will be encodable.

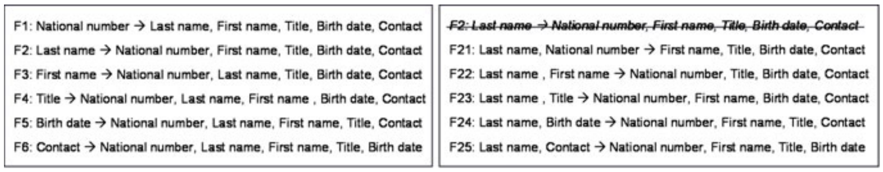
4.2 Initialisation

Before beginning the interaction with the end-users, we start by initialising an empty set of tuples and defining the initial sets of validated, potential and discarded constraints and dependencies for each entity type associated with a given form. The *validated* constraints are initially the same than the previously *declared* constraints. From these initial validated constraints, the *potential* domains of value, cardinalities and existence constraints are initialised using *induction*. Typically, if a given component is considered optional so far, it could actually be mandatory if there is no tuple with this component empty (whereas the opposite is not possible). Similarly, an attribute declared as textual could be of any other type, while an attribute declared as real could only be restricted to the type integer. In the same way, any subset of optional components for which no existence constraint has been declared should be submitted to existence constraint elicitation. Consider for instance that the attribute `Zip code` has been declared optional and textual in Fig. 2. Further examination should therefore check whether this attribute is not mandatory and whether its value domain is not restricted to integers, reals, dates, ...

Regarding functional dependencies, the ideal process should lead us to build a set of data samples and dependencies so that each entity type of the underlying conceptual schema becomes an *Armstrong relation*. Reaching such a state is

[2] Note that the GER notation uses the *participation* interpretation rather than the *look-across* interpretation (as in UML). In the given example, the notation therefore indicates that same contact details may apply to more than one person.

obviously not trivial *per se*, and these principles are here inapplicable due to the requirement of user involvement. However, we can try to near it by progressively narrowing the functional dependencies. Since the number of possible functional dependencies for each entity types can be very high, we start from the set of *strongest* dependencies, through which each component of a given entity type determines the other components. For instance, the form of Fig. 2 induces the initial functional dependencies F_1, F_2, F_3, F_4, F_5 and F_6 of Fig. 3. From these dependencies, we will be able to recursively generate *weaker* functional dependencies to cover all the existing ones, by progressively reducing the right-hand sides and enlarging the left-hand sides. The objective is to favour functional dependencies with minimal left-hand sides and maximal right-hand sides.



F1: National number → Last name, First name, Title, Birth date, Contact
F2: Last name → National number, First name, Title, Birth date, Contact
F3: First name → National number, Last name, Title, Birth date, Contact
F4: Title → National number, Last name, First name , Birth date, Contact
F5: Birth date → National number, Last name, First name, Title, Contact
F6: Contact → National number, Last name, First name, Title, Birth date

~~F2: Last name → National number, First name, Title, Birth date, Contact~~
F21: Last name, National number → First name, Title, Birth date, Contact
F22: Last name , First name → National number, Title, Birth date, Contact
F23: Last name , Title → National number, First name, Birth date, Contact
F24: Last name, Birth date → National number, First name, Title, Contact
F25: Last name, Contact → National number, First name, Title, Birth date

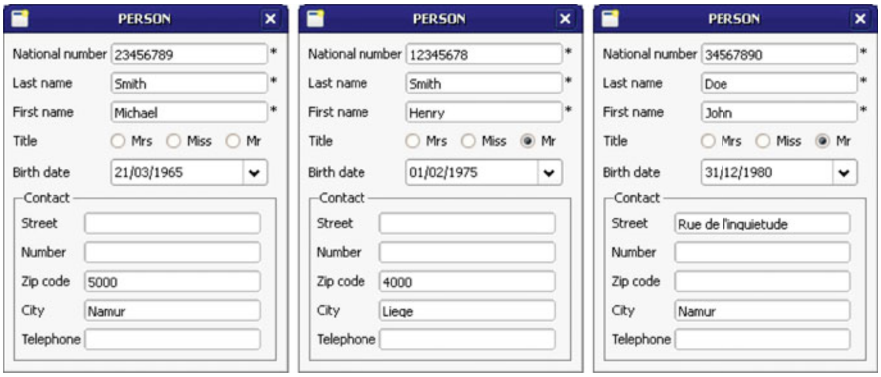
Fig. 3. The initial functional dependencies F_1, F_2, F_3, F_4, F_5 and F_6 for form of Fig. 2, as well as alternatives for F_2

Finally, potential *unique constraints* are induced from validated and potential functional dependencies, using the fact that the left-hand side of a given functional dependency $f : X \rightarrow Y$ is a potential identifier for an entity type having the set of components $\mathcal{C} = X \cup Y$.

4.3 Analysing New Data Samples to Suggest Constraints and Dependencies

Once the sets of constraints and dependencies have been initialised, we take advantage of user input to acquire data samples that will progressively reduce the set of potential constraints and dependencies. To be consistent with the previously validated constraints and dependencies, any new tuple must respect the latter to be accepted. Once a new tuple is acceptable, we proceed with its analysis to determine which potential constraints and dependencies do not hold any more. The invalidated constraints are discarded, while the invalidated functional dependencies are replaced by alternative dependencies. Let us explicit this process for each type of constraint and dependency when adding a new valid tuple. Fig. 4 illustrates three data samples that could be encoded for the form of Fig. 2, and Fig. 5 illustrated the underlying relation `Person` after the acquisition of these data samples. Despite the apparent structure of attribute `Contact`, this relation is in 1NF. Indeed, the compound value must be considered as a whole whose unique goal is to reference a target tuple in the `CONTACT` table.

First of all, discarding the potential *domains of value* and *cardinalities* that do not hold any more is relatively straightforward, since it consists in removing



PERSON

National number	23456789 *
Last name	Smith *
First name	Michael *
Title	Mrs Miss Mr
Birth date	21/03/1965
Contact	
Street	
Number	
Zip code	5000
City	Namur
Telephone	

PERSON

National number	12345678 *
Last name	Smith *
First name	Henry *
Title	Mrs Miss [✓ Mr]
Birth date	01/02/1975
Contact	
Street	
Number	
Zip code	4000
City	Liege
Telephone	

PERSON

National number	34567890 *
Last name	Doe *
First name	John *
Title	Mrs Miss [✓ Mr]
Birth date	31/12/1980
Contact	
Street	Rue de l'inquietude
Number	
Zip code	
City	Namur
Telephone	

Fig. 4. Three data samples for the form of Fig. 2

	PERSON					
tuples	National Number	Last Name	First Name	Title	Birth date	Contact
t1	23456789	Smith	Michael	Ø	21/03/1965	< Ø, Ø, 5000, Namur, Ø >
t2	12345678	Smith	Henry	Mr	01/02/1975	< Ø, Ø, 4000, Liege, Ø >
t3	34567890	Doe	John	Mr	31/12/1980	< Rue de l'inquiétude, Ø, Ø, Namur, Ø >

Fig. 5. The relation `Person` after the acquisition of the data samples of Fig. 4. The compound value of attribute `Contact` must be considered as a whole whose unique goal is to reference a target tuple in the corresponding `CONTACT` table.

the constraints with which the tuple does not agree. Regarding the cardinalities, we remove the possible mandatory constraints for components that are empty, and we remove the value type constraints that are not compatible with the value provided for each attribute and replace the value size if the provided value is longer. For instance, adding the first data sample confirms that the `Title` is definitely optional, while the `Birth date` potentially remains a mandatory attribute. The `Zip Code` of a `Contact` can now only be validated as integer, real or textual. The second data sample still supports `Birth date` being a possibly mandatory attribute.

Secondly, discarding the potential *existence constraints* that do not stand any more also consists in removing the constraints with which the tuple does not agree. Consequently, coexistence constraints are removed if their set of components is different from the set of non empty optional components of the tuple. Exactly-one, exclusion and at-least-one constraints are respectively removed if there is not one and only one, more than one or less than one of their components that is not null among the set of non empty optional components of the tuple. Finally, we remove all the implication constraints for optional components if the suggested prerequisite components are not part of the non empty components of the tuple. The first data sample for instance suggests that there could be at-least one, at-most one or exactly one value of `Title` or `Birth date`, or that the former could require the latter (implication). The second data sample implies that the only remaining potential configurations for `Title` or a `Birth date` is at-least-one, or that the former requires the latter.

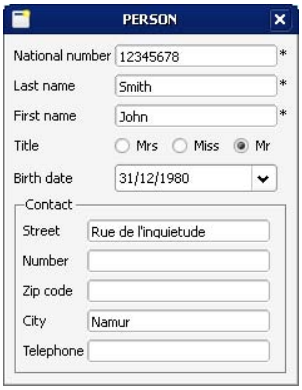
We also analyse each potential *functional dependency* to check if there is a conflictual tuple among the previously provided tuples, i.e. if an existing tuple has the same left-hand side but a different right-hand side when considering the components of the functional dependency. If such a conflictual tuple exists, the functional dependency is discarded and alternatives are recursively generated. First of all, this implies that the right-hand side may be too large with respect to left-hand side, and we therefore consider smaller right-hand sides by removing a component. The removed component may be purely dismissed, or added to the left-hand side to consequently generate two alternatives per component. For instance, the first data sample doesn't jeopardize the potential functional dependencies, but the second data sample discards the FD F_2 and generates the alternatives F_{21}, F_{22}, F_{23}, F_{24} and F_{25} of Fig. 3. The second data sample then discards the FD F_4 and generates its subsequent alternatives.

Understanding the implications of a functional dependency is not always trivial and easy to grasp. Presenting the end-users with automatically generated data samples that would contradict the validity of existing functional dependencies therefore helps them to visualise the relevance of these dependencies, while reducing the number of tuples that they would need to provide by themselves. As we can observe, a tuple t is actually problematic for the functional dependency $f : X \rightarrow Y$ and the existing set of tuples $\mathcal{T}$ if $\exists\, t' \in \mathcal{T} : t'[X] = t[X] \wedge t'[Y] \neq t[Y]$. If we already have several tuples in the tuples set of a given entity type, we generate problematic data samples for a given dependency by putting together previously provided data samples. Accepting such a generated data sample would imply discarding the associated functional dependency and generate alternatives. For instance, considering the functional dependency F_{23}, we generate the problematic tuple illustrated in Fig. 6 from the composition of the second and third data samples of Fig. 4.

Finally, potential *unique constraints* are again induced from validated and potential functional dependencies, using the same principle than during the initialisation of the process, i.e. the left-hand side of a functional dependency involving all the components of a given entity type is a potential identifier for that entity type.

4.4 Acquiring Constraints and Dependencies

Another way to take advantage of user input is to directly acquire validated or discarded constraints and dependencies, whenever they are trivial and easy to express for the participants, and to invite them to arbitrate the potential constraints and dependencies that could be suggested after the acquisition of multiple data samples. The end-users should indeed be able to directly specify validated or discarded constraints and dependencies, even without looking at possible suggestions. To be accepted as validated, a given constraint or



PERSON

National number	12345678	*
Last name	Smith	*
First name	John	*
Title	Mrs Miss [✓ Mr]	
Birth date	31/12/1980	

Contact

Street	Rue de l'inquietude
Number	
Zip code	
City	Namur
Telephone	

Fig. 6. A problematic data sample for the FD `Title` → `National number, Last name, First name, Birth date, Contact`, given the valid data samples of Fig. 4

dependency must be satisfied by the existing set of tuples associated with the considered entity type.

Alternatively, the participants can also take advantage of the potential constraints and dependencies to arbitrate them, i.e. to validate or discard them. The advantages of this approach are that the participants do not have to imagine all the possible constraints and dependencies for each entity type, and that we directly know that each candidate constraint or dependency is currently potential for the given entity type. One can suspect that validating or discarding a constraint or a dependency may impact on the constraints or dependencies of other types. Such a correlation actually exists between functional dependencies and unique constraints. Indeed, discarding a potential functional dependency may change the potential unique constraints, whereas validating a unique constraint automatically validates its underlying functional dependency and discards others. When these cases occur, the relevant sets must therefore be updated.

Besides, it obviously appears that the number of suggested constraints and dependencies can eventually become very high. It is therefore crucial to organise these suggestions in an accessible fashion, so that the end-users do not feel overwhelmed. Besides, this underlines the importance of the analyst to guide the end-users through this collaborative process, by assessing the relevance of these suggestions. This observation is especially true regarding the elicitation of the functional dependencies, since the number of suggestions can increase dramatically.

We therefore propose to filter the potential functional dependencies in order to limit the number of relevant suggestions, while privileging the "stronger" functional dependencies (i.e. the dependencies with smaller left-hand side and larger right-hand side, as previously explained). For this purpose, we therefore propose to "hide" dependencies that can be obtained from other potential dependencies using *Armstrong's axioms*, which are a set of inference rules used to infer all the functional dependencies on a relation [19]. Hiding these functional dependencies does not mean discarding them. Indeed, they are still potential, and may eventually become visible again with the progressive arbitration of the other dependencies. Still, we observed that this filtering helps keeping the focus of the end-users during this elicitation process.

5 Evaluation

To experiment and evaluate the RAINBOW approach, a validation protocol was defined based on the *Participant-Observer* principles to monitor the use of the RAINBOW approach, and the *Brainstorming/Focus group* principles to analyse the resulting conceptual schemas, as defined in [20]. This protocol was used for a first series of experiments where pairs of end-users and analysts were asked to jointly define the conceptual schema of a future information system, including constraints and dependencies, using the RAINBOW approach and its tool support.

For each project, the first task consisted in preparing the experimentation by defining the subject based on real-life concerns of the end-users, then training the participants to understand the method and to use the tools. Secondly, the end-users and analysts were asked to apply the approach on their project and focus on the five first phases, while observers took notes. In particular, for the *Nurture* phase which dealt with the elicitation of constraints and dependencies, the participants were asked to progressively provide data samples and constraints, while arbitrating the candidate constraints suggestions. Thirdly, the observations on the efficiency of the approach were analysed, and finally, the quality of the produced schemas was debated, taking in account schemas that were designed by the analysts independently of the approach.

The analysis of these experiments notably highlighted that the RAINBOW approach and tool support did help end-users and analysts to communicate static data requirements to each other, inclusive of constraints and dependencies. Though all the requirements could not be expressed through the toolkit, the latter did serve as a basis for discussion and modifications. Since the validation aspect of the proposed approach cannot be addressed more extensively in this paper, the interested reader may refer to [21] for further details on the validation process and methodology.

6 Conclusion

In this paper, we extend the user-oriented RAINBOW approach presented in [2], and describe how it can be used to interactively elicit database constraints and dependencies, and more specifically domains of values, optionality, existence constraints, identifiers and functional dependencies. The process, inspired by the principles of Armstrong relations, uses form-based interfaces that were previously drawn by the end-users themselves in order to let them provide a limited set of data samples that will restrict the potential implicit constraints of the underlying conceptual schema. Conversely, end-users are invited to arbitrate potential constraints that will in turn restrict the tuples that can be encoded. Such a process prevents the development of unsatisfiable systems of constraints and dependencies, since such set of constraints will never accept the introduction of new tuples. Whereas usual dependency discovery approaches rely on extensive data sets, this specific *modus operandi* is particularly adapted when engineering information systems with no legacy data samples available, or when their re-encoding would be too expensive. The application of this approach to different case studies have proved that such intensive end-user involvement with interactive support is particularly fruitful and sustainable, while merely providing, without support, significant amount of data samples is a particularly tedious and time-consuming process, and in most situations, unrealistic. Besides, manipulating form-based interfaces to encode data samples leads to directly expressing trivial constraints, while inducing further discussion and reflection on their underlying conceptual schema. Though this approach relies on a set of pre-existing form-based interfaces, its principles are easily generalisable to any given conceptual schema. Indeed, the constructs of the schema can first be transformed to comply with the structures used in the RAINBOW approach [4]. Subsequently, a set of form-based interfaces can then be generated from this transformed schema [22], hence enabling the encoding of data samples and the application of the proposed approach.

References

1. Rosson, M.B., Carroll, J.M.: Usability Engineering: Scenario-Based Development of Human-Computer Interaction (Interactive Technologies). Morgan Kaufmann, San Diego (2001)
2. Ramdoyal, R., Cleve, A., Hainaut, J.-L.: Reverse engineering user interfaces for interactive database conceptual analysis. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 332–347. Springer, Heidelberg (2010)
3. Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM 13(6), 377–387 (1970)
4. 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)
5. Ram, S.: Deriving functional dependencies from the entity-relationship model. Communications of the ACM 38(9), 95–107 (1995)
6. Lopes, S., Petit, J.-M., Lakhal, L.: Functional and approximate dependency mining: database and FCA points of view. Journal of Experimental and Theoretical Artificial Intelligence (JETAI) 14(2-3), 93–114 (2002)
7. Yao, H., Hamilton, H.J.: Mining functional dependencies from data. Data Mining and Knowledge Discovery 16(2), 197–219 (2008)
8. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE: An efficient algorithm for discovering functional and approximate dependencies. Computer Journal 42(2), 100–111 (1999)
9. Novelli, N., Cicchetti, R.: FUN: An efficient algorithm for mining functional and embedded dependencies. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 189–203. Springer, Heidelberg (2000)

10. Lopes, S., Petit, J.-M., Lakhal, L.: Efficient discovery of functional dependencies and armstrong relations. In: Zaniolo, C., Grust, T., Scholl, M.H., Lockemann, P.C. (eds.) EDBT 2000. LNCS, vol. 1777, pp. 350–364. Springer, Heidelberg (2000)
11. Wyss, C.M., Giannella, C., Robertson, E.L.: Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In: Kambayashi, Y., Winiwarter, W., Arikawa, M. (eds.) DaWaK 2001. LNCS, vol. 2114, pp. 101–110. Springer, Heidelberg (2001)
12. Priss, U.: Establishing connections between formal concept analysis and relational databases. In: Common Semantics for Sharing Knowledge: Contributions to ICCS 2005, pp. 132–145 (2005)
13. Baixeries, J.: A formal concept analysis framework to mine functional dependencies. In: Proceeding of Mathematical Methods for Learning 2004: Advances in Data Mining and Knowledge Discovery (2004)
14. Correia, J.H.: Relational scaling and databases. In: Proceedings of the 10th International Conference on Conceptual Structures (ICCS 2002), Borovets, Bulgaria, July 15-19, pp. 62–76 (2002)
15. Rancz, K.T.J., Varga, V.: A method for mining functional dependencies in relational database design using FCA. Studia Universitatis Babes-Bolyai Cluj-Napoca, Informatica LIII(1), 17–28 (2008)
16. Rancz, K.T.J., Varga, V., Puskas, J.: A software tool for data analysis based on formal concept analysis. Studia Universitatis Babes-Bolyai Cluj-Napoca, Informatica LIII(2), 67–78 (2008)
17. Flory, A.: Bases de données: conception et réalisation. In: ECONOMICA, Paris (1982)
18. Lopes, S., Petit, J.-M., Lakhal, L.: Efficient discovery of functional dependencies and armstrong relations. In: Zaniolo, C., Grust, T., Scholl, M.H., Lockemann, P.C. (eds.) EDBT 2000. LNCS, vol. 1777, pp. 350–364. Springer, Heidelberg (2000)
19. Armstrong, W.W.: Dependency structures of data base relationships. In: IFIP Congress, pp. 580–583 (1974)
20. Singer, J., Sim, S.E., Lethbridge, T.C.: Software engineering data collection for field studies. In: Shull, F., Singer, J., Sjøberg, D.I. (eds.) Guide to Advanced Empirical Software Engineering, pp. 9–34. Springer, Heidelberg (2008)
21. Ramdoyal, R.: Reverse Engineering User-Drawn Form-Based Interfaces for Interactive Database Conceptual Analysis. PhD thesis, University of Namur, Namur, Belgium, (December 2010) Electronic version available from `http://www.info.fundp.ac.be/libd/rainbow`
22. Pizano, A., Shirota, Y., Iizawa, A.: Automatic generation of graphical user interfaces for interactive database applications. In: CIKM 1993: Proceedings of the Second International Conference on Information and Knowledge Management, pp. 344–355. ACM, New York (1993)