DOCTOR OF SCIENCES

Reverse engineering: user-drawn form-based interfaces for interactive database conceptual analysis: the rainbow approach

Ramdoyal, Ravi

Award date:
2010

Awarding institution:
University of Namur

Reverse Engineering

User-Drawn Form-Based Interfaces

for Interactive Database Conceptual Analysis

Ravi R.C. Ramdoyal

Doctor of Philosophy Dissertation
Namur, 2010

Laboratory of Database Applications Engineering
Faculty of Computer Science • PReCISE Research Centre • University of Namur

©Presses Universitaires de Namur
Rempart de la Vierge, 13
B - 5000 Namur (Belgique)

Imprimé en Belgique
ISBN : 978-2-87037-696-6
Dépôt légal: D/2010/1881/42

Doctoral Committee

Prof. Jean-Marie Jacquet (President)
Faculty of Computer Science
University of Namur (Belgium)

Prof. Jean-Luc Hainaut (Promotor)
Faculty of Computer Science
University of Namur (Belgium)

Prof. Patrick Heymans (Internal Reviewer)
Faculty of Computer Science
University of Namur (Belgium)

Prof. Monique Noirhomme-Fraiture (Internal Reviewer)
Faculty of Computer Science
University of Namur (Belgium)

Prof. Jean Vanderdonckt (External Reviewer)
Faculty of Economical, Social and Political Sciences
University of Louvain-la-Neuve (Belgium)

Prof. Óscar Pastor López (External Reviewer)
Centro de Investigación en Métodos de Producción de Software (ProS)
Universidad Politécnica de Valencia (Spain)

The public PhD defence was held on the 15th of December 2010 at the University of Namur.

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 are formalised into a conceptual schema of the application domain, that has proved difficult to validate, especially since the graphical representations of data models have shown understandability limitations from the end-users standpoint. On the other hand, electronic forms seem to be more natural and intuitive to express data requirements for laymen. Besides, the necessity to associate end-users of a future system with its specification and development steps has long been advocated.

In this doctoral research, we consequently explore the possible reverse engineering of user-drawn form-based interfaces to perform an interactive database conceptual analysis, and subsequently present the tool-supported RAINBOW approach resulting from this investigation. This user-oriented approach 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.

Keywords: Software Engineering, Requirements Engineering, Data Modelling, Database Forward Engineering, Database Reverse Engineering, Human-Computer Interfaces, Prototyping, Participatory Design.

Résumé

La première étape de la plupart des méthodologies de développement de bases de données se concentre notamment sur l'élicitation des besoins des utilisateurs, à partir de sources diverses telles que des interviews et de la documentation existante. Ces besoins sont formalisés par le biais d'un schéma conceptuel représentant le domaine d'application. En pratique, la validation de tels schémas s'avère difficile, étant donné que la compréhension de leur formalisme, même à l'aide de représentations graphiques, est loin d'être triviale. Parallèlement, il apparait que les formulaires électroniques semblent plus naturels et intuitifs pour permettre l'expression de tels besoins par des non experts. Par ailleurs, l'implication des utilisateurs finaux dans la spécification et le développement d'un futur système fait désormais partie des bonnes pratiques de l'Ingénierie Logicielle.

Dans cette recherche doctorale, nous explorons dès lors la possible rétro-ingénierie d'interfaces de type formulaire dessinées par des utilisateurs finaux, afin de permettre une analyse conceptuelle interactive. Le résultat de cette investigation est l'établissement de l'approche RAINBOW et de son support logiciel. Cette approche orientée utilisateur se base sur l'adaptation et l'intégration de principes et de techniques provenant de diverses disciplines, notamment l'Ingénierie et la Rétro-ingénierie des Bases de Données, le Prototypage et le Design Collaboratif.

Mots-clef: Ingénierie Logicielle, Ingénierie des Besoins, Modélisation des Données, Ingénierie des Bases de Données, Rétro-ingénierie des Bases de Données, Interfaces Homme-Machine, Prototypage, Design Collaboratif.

Acknowledgements

Let us be grateful to people who make us happy; they are the charming gardeners who make our souls blossom.
Marcel Proust

When this adventure started a few years ago, little did I expect that leading a doctoral research would be such a unique blend of sweet and sour paradoxes, with its faire share of upsides and downsides. As it has already been said elsewhere, on the one hand, being a PhD student is a lonely, laborious and worrisome experience. But on the other hand, it offers so many positive boons and rewards that I would rather only remember the latter. Indeed, during this timespan, I notably got the chance to inquire about many different and exciting topics and technologies, while meeting so many educated, interesting and caring people, always ready to provide their help and cheer you up.

Among these persons, I would first of all like to express my gratitude and appreciation to my supervisor, Prof. Jean-Luc Hainaut, who notably gave me the opportunity to embrace the challenge of pursuing a PhD. Thank you for teaching me to become more autonomous and proactive in my research through your confidence and the latitude you granted me during these years. Thank you for your time, your precious advices and your insight on my work. Not forgetting the chance of being chosen as teaching assistant. Teaching and supervising students at the Faculty of Computer Science was indeed a fulfilling and enjoyable experience as well.

Next, I would like to thank the members of my jury, Prof. Monique Noirhomme-Fraiture, Prof. Patrick Heymans, Prof. Jean-Marie Jacquet, Prof. Jean Vanderdonckt and Prof. Óscar Pastor López for accepting to participate in the evaluation of my work. It was an honour and a pleasure to discuss my

research with you, especially given your encouragements and valuable feedback. I am hopefully looking forward to keep in touch and eventually collaborate with you on future research projects.

Regarding the preliminary case studies that were led for the validation of this research, I would like to warmly thank my anonymous participants, as well as Patrick for his acute suggestions and recommendations. Thank you so much for your time, your enthusiasm and your constructive input. Beyond the great importance of your collaboration regarding my research, it was pleasure to simply be able to get together. Besides, I also want to thank the other people who indirectly contributed and influenced this research, either as part of the ReQuest project or as part of the DB-Main team.

I am also very grateful to Giz, Anthony and Benjamin, my fellow doctor predecessors, for showing me the right direction. Their perseverance and dedicated hard work were a true inspiration in times where I was in the darkness and uncertainty of my doctoral journey. Thank you also for your understanding solicitude and your cheering during my work.

Likewise, I want to thank Alain and Benoit for being the daily and supportive witnesses of my doctoral progress. Thank you for the humorous, relaxing and necessary time-outs that we occasionally shared with Flora and Abdelkader, and which often gave me the necessary boost to resolutely go back to work. As you are yourself nearing the end of your doctoral path, you can count on me to root for you and wish you only the very best!

I would then like to thank my past and present colleagues of the Laboratory of Database Application Engineering, Anne-France, Rokia, Sophea, Virginie, Anthony, Eric, Jean-Roch, Jonathan, Julien, Vincent and Yannis. Thank you for your concern, your support, and all these good times that undoubtedly made us more than simple co-workers.

I would also like to thank our top-notch secretaries, Anne-Marie, Babette, Gyselle and Isabelle for making the organisation and practical progress of my work so much easier. Thank you also for your kindness and consideration, beyond the mere doctoral preoccupations.

More generally, I would like to thank all the other friends, colleagues and unsuspected strangers who rooted for me during my doctoral tenure. There are too many of them to mention them all, but I am truly grateful for all the consideration and support that I received. Thank you.

Last but not least, I wish to thank the members of my family for their unconditional love and support. In particular, my parents gave and taught me so much that I am eternally grateful to them. Dear Mataji and Pitaji, thank you for everything. I am also grateful to my family-in-law, for their warmth, kindness and impulse. Thank you Bernadette, Pierre, Caroline, Corentin, Jean-

Baptiste, Véronique and Elisabeth for your upbeat presence. And finally, I would like to thank my wife, Emy for her endless love, understanding and encouragements. Thank you for all these precious moments together, and for giving me the strength to carry on through enduring times. Thank you for being there no matter what, especially during this last year which has been incredibly rich in changes and challenges. At the end of the day, when I look at our beautiful little Louise, I guess that we made it through pretty well, and that there are many more blessings to come. Cheers!

This dissertation is dedicated in loving memory
of Nani, my grand-mother, for always
pushing me to work hard
and stay the course.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

Chapter 1

Introduction

In the realm of Software Engineering, there is a dark and inhospitable land, at the crossroads of Requirements Engineering, Database Engineering and Human-Computer Interfaces. In that mysterious place, people explore and experiment ways to combine these disciplines in order to provide better methods and means to lay the foundations of an efficient and reliable software development cycle. This is the place where our journey begins.

Requirements Engineering is a key step of Software engineering, since it defines the necessary specifications for further analysis, design and development. Within this process, 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. Since long, conceptual schemas have proved to be difficult to validate by laymen, while traditional database requirements elicitation techniques, such as the analysis of corporate documents and interviews of stakeholders, usually do not actively and interactively involve end-users in the overall specification and development of the database.

Still, the necessity to associate end-users of the future system with its specification and development steps has long been advocated. In particular, the process of eliciting static data requirements should make end-users feel more involved and give them intuitive and expressive means to convey their requirements to the analysts. Conversely, analysts should also be able to capture and validate these requirements by discussing them with the end-users. Yet, most users are fortuitously quite able to deal with complex data structures that are expressed through more natural and intuitive layouts such as electronic forms.

In order to facilitate this communication, we therefore investigate the possible reverse engineering of user-drawn form-based interfaces to perform an interactive database conceptual analysis, and propose an approach to elicit and validate static database requirements based on end-users involvement through interactive prototyping and the adaptation of techniques coming from various fields of study. Our dissertation will hence be organised as follows.

Part I presents the context of our research, and more precisely the aspects of Requirements Engineering, Database Engineering and Prototyping that led us to inquire about a new approach to elicit and validate database requirements. The challenges implied by the perspective of integrating these different disciplines and techniques into an integrated user-oriented approach are then exposed. Part II subsequently presents the tool-supported RAINBOW approach for reverse engineering user-drawn form-based interfaces in order to perform an interactive database conceptual analysis. Part III then presents the experimentation that was led to validate this approach. Part IV consequently discusses the specificities and merits of the approach, as well as limits and possible improvements that could be considered, before concluding this dissertation.

Part I

Problem Statement

In this part of the dissertation, we address the general background of this doctoral investigation. For this purpose, Chapter 2 focuses on its research context, and highlights different aspects of Requirements Engineering, Data Modelling and Prototyping that that led us to inquire about a new approach to elicit and validate database requirements, based on the reverse engineering of user-drawn form-based artefacts. Chapter 3 then presents the state of the art for this field of research, as well as the resulting research perspectives, before introducing a series of essential research questions that need to be addressed in order to formulate a comprehensive proposal.

Chapter 2

Research context

In this part of the dissertation, we present the elements that led us to inquire about a new approach to elicit and validate database requirements.

First of all, Section 2.1 recalls that forty years ago, the Software Crisis already paved the way for more formal engineering approaches in Software Design. While the overall process gained in reliability, another crisis still occurred in the nineties, putting in light the need for a requirements analysis phase, which should be led with collaboration of all the stakeholders.

The critical aspects and challenges of this fundamental phase, which should lay the foundations of the remainder of any Software Engineering process, are then presented. While trying to get a better insight of the client needs, it appears that Requirements Engineering still calls for new approaches to involve more intimately future end-users in the definition and validation of the requirements.

Within this context, Section 2.2 introduces the particular case of requirements elicitation in Database Engineering. We will see that various techniques exist to acquire data requirements, but usually, they precisely do not actively and interactively involve end-users. We then introduce Database Reverse Engineering, as well as the Generic Entity-Relationship model and the principles of the transformational approach.

Section 2.3 then presents the Requirements Engineering technique called Prototyping, which can be used in order to bridge the gap between the various stakeholders of a software engineering project.

2.1 Requirements Engineering

In this section, we briefly recall the main crises that hit the realm of software development these last decades, and how they respectively led to the advent of *Software Engineering* and *Requirements Engineering* (RE). We then present the core activities and main challenges of Requirements Engineering, as well as the areas where improvements can still be made.

2.1.1 Lessons learned from the Software Crises

The different software crisis

If we go back in time, the difficulties of developing large software systems appeared during the sixties and seventies. This is when the term of *software crisis* emerged [Osmundson et al., 2003]. This crisis occurred because the techniques used to build small software systems did not scale up. This resulted in various failures, cost overruns and long lead-times. By the time the information systems projects were completed, it was not surprising to record that the initial organisational requirements had already changed.

Software Engineering was originally a response to this crisis, relying on the types of theoretical foundations and practical disciplines, which are traditional in the established branches of engineering [Naur et al., 1976]. The main objectives were to get control of the software development process and to improve software performance and software reliability. This led to new approaches of software engineering management and new techniques of software development.

However, using formal methods was not enough to prevent the crisis from re-emerging during the nineties, mainly because the demands placed on the software engineering community, such as productivity, flexibility, robustness and quality, have increased dramatically [Mehandjiev et al., 2002], while our dependency on large software systems has also been rising [PITAC, 1999].

Risk factors

As a matter of fact, only a few software projects are completed on-time and on-budget, while the majority of them are completed and operational. However, the latter often do not come free of defect: they carry budget and/or time overrun, missing and/or reduced functionalities [Standish, 1995]. The Standish Group surveyed IT executive managers for their opinion about which factor could lead to successful, challenged or cancelled projects. The major *success criteria* were user involvement, as well as executive management support and clear statement of requirements. The major *struggle criteria* were the lack

of user input, the incomplete requirements and specifications as well as the modification of requirements and specifications. Finally, the major *cancellation criteria* were incomplete requirements, the lack of user involvement and the lack of resources. In the other cases, due to the frequent changes, the budget may be largely exceeded.

As a result, the Standish Group identified the three biggest contributors to project success as user involvement, clear business objectives and executive support [Standish, 1999]. Since then, their importance have somehow evolved, but they still are key factors for project achievement [Standish, 2001]. Actually, McConnell classifies them as part of 12 best influences on software engineering [McConnell, 2000].

It seems therefore obvious that the step in software development should consist in acquiring reliable requirements and specifications. Since the following analysis and design phases cannot be led rigorously if the output of this initial phase is not complete and coherent, it appears that the requirements phase is fundamental, and should be led with collaboration of the stakeholders.

2.1.2 An overview of Requirements Engineering

Definition

Among the various definitions of *Requirements Engineering*, let us put forward the one formulated by [Nuseibeh and Easterbrook, 2000]:

> "Broadly speaking, software systems Requirements Engineering (RE) is the process of discovering [the purpose for which a software system is intended], by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation".

Intuitively, this collaborative effort is a multi-disciplinary human-centred process that involves different activities producing various outputs while implying a number of challenges. Though there is little uniformity in the terminology and ordering of the classes of activities that make up Requirements Engineering, there seems to be at least an agreement on the following core tasks [Davis and Zowghi, 2006]: elicitation, analysis and documentation modelling, validating specifications, managing the evolution of requirements.

Eliciting requirements

This complex activity is the starting point of any requirements engineering process, as it should lead to understanding the context of the software engineering project, as well as the expectations that need to be answered.

Within this context, the first step consists in identifying the *environment* and especially the *stakeholders*, which may potentially be anyone whose job or influence will be altered within the organisation, as well as anyone involved in the process of sharing information and participating in the elaboration and set up of the final solution [Dix et al., 1998]. Among possible stakeholders [Sharp et al., 1999], we can therefore notably identify:

- The *buyers*, which are the individuals that are responsible for contracting and/or paying for the software system. In the end, they are the ones who decide whether to go for the proposed solution or not;
- The *managers*, which are members of the hierarchy whose decisional power can be altered through the process of the software system project. As pivotal elements of the organisation, they can influence the outcome of the whole process;
- The *domain and application experts*, who provide domain knowledge and supply a more detailed understanding of the problem, of the business rules and conventions, as well as peculiar "modi operandi";
- The *end-users*, which are the individuals who should ultimately install, operate, and use the software. There can be different classes of users, according to the type of tasks they are meant to perform through the software, and their acceptance and handling of the software is crucial to determine its success;
- The *requirements engineers*, which are the individuals who are responsible for the identification, formalisation and documentation of the requirements. In order to accomplish that, they must mediate between all the other stakeholders;
- The *software engineers* and *developers*, which are the individuals who provide expertise on software design techniques and technologies;
- The *testers*, who test the system to perform some predefined set of tasks and compare the execution to expected results. They may (but must not necessarily) be end-users.

In addition to these human beings, factors such as physical, organisational and legislation environments may also be taken in account [Vries et al., 2003]. Once the context is defined, the next step consists is determining the *problem* and drawing its *boundaries* consequently. This implies delimiting the objectives

of the project and the vision of the stakeholders in order to determine what subset of the requirements should actually be addressed given the constraining budgets and schedules. While bearing these aspects in mind, it is also necessary to clearly identify the *goals* and the *tasks*. Delineating these elements calls for various and combinable elicitation techniques [Goguen and Linde, 1993; Sommerville and Kotonya, 1998; Nuseibeh and Easterbrook, 2000], among which the most usual are:

- *traditional techniques*, such as existing corporate documents analysis, questionnaires, interviews;
- *group elicitation techniques*, such as brainstorming and focus groups;
- *early development techniques*, such as Prototyping and Rapid and/or Joint Application Development (RAD/JAD);
- *observation techniques*, typically *contextual approaches* such as stakeholder observation or *cognitive techniques* such as protocol analysis.

If thoroughly led, this task should produce a fair amount of information that subsequently needs to be processed.

Analysing and modelling requirements into specifications

Indeed, the extracted information must be analysed in detail by the requirements engineers to get a better overview of the project, understand its ins and outs while identifying priority levels (e.g. mandatory and "measurable" needs versus optional and "vague" wishes) and satisfiable demands. The primary output of this cogitation is a *Software Requirement Specification* (SRS), which should typically address:

- *Enterprise Modelling*, which presents the organisation's structure, business rules, tasks and goals;
- *Data Modelling*, which defines the application domain (possibly reusing existing domain-specific models) and presents the information that needs to be manipulated by the system in terms of structured data;
- *Behaviour Modelling*, which defines the dynamic and functional behaviour of the users and the system;
- *Non-Functional Requirements*, which deal with the constraints imposed to the system and its expected qualities, notably regarding its execution and evolution.

Besides, the specification produced are vital to begin the design and implementation phases, which implies that the complete Software Requirements

Specification should possess the following characteristics in order to produce high quality software [IEEE, 1998]:

- *Correct*: Every requirement stated in the SRS is one that the software shall meet;
- *Unambiguous*: Every requirements stated in the SRS has one and only one logical interpretation;
- *Complete*: The SRS includes all significant requirements, all realisable classes of input data as well as the labels and references to all figures, tables and diagrams. No requirements or necessary information should be missing;
- *Consistent*: No subset of individual requirements stated in the SRS conflict with other individual requirements. Disagreements among requirements must be resolved before development can proceed;
- *Verifiable*: For every requirement stated in the SRS, there exists some finite cost-effective process with which a person or machine can check that the software meets the requirements;
- *Modifiable*: The entire SRS has a style and structure such that any changes to the requirements can be made easily, completely, and consistently which retaining the structure and style;
- *Traceable*: For every requirement stated in the SRS, the origin is clearly stated and it is possible to reference each requirement in future developments;
- *Concise*: Unnecessary literature and redundancies should be avoided;
- *Testable*: Pass/fail or quantitative assessment criteria can be derived from the specification itself and/or referenced information.

To meet such expectations, the Requirements Engineering can take advantage of a wide variety of modelling approaches and languages, among which the *Unified Modelling Language* (UML) [OMG, 2007] has become a standard, thanks to its rich set of views on software systems: use cases, class diagrams, sequence diagrams, state charts, activity diagrams, etc.

Sharing and validating requirements

The Software Requirement Specification should also be packaged and *documented* in a way allowing the *communication* between any stakeholders, in order to allow *validation* and *agreement*. However, all the stakeholders do not necessarily understand easily formal specifications, and in parallel, computer analysts and developers may struggle to grasp the client's problem and the

application area. Therefore, this document should aim at bridging communication gaps in order to typically ease:

- *validation*: the stakeholders can evaluate if their own needs are expressed accurately, and even perhaps reconsider them;
- *correction*: in particular, omissions, inconsistencies and misunderstandings can be quickly and easily identified;
- *forecast*: a realistic estimation can be established for the performing cost, schedule, and necessary resources relative to the developmental effort;
- *evaluation criteria*: it can provide a baseline for verification and validation of the software;
- *future reference*: it can provide rigorous specifications and documentation for the design and implementation phase;
- *contractibility*: and ultimately a safe and sound contractual agreement can be established between the different parties.

Managing changes to requirements

Finally, since the elaboration of the Software Requirement Specification is clearly iterative, the requirements should be traceable over time, and easy to edit and monitor. This could for instance include modifying and adding new requirements to clarify and better comply with the stakeholders needs, or scrubbing requirements to meet the budget and planning.

2.1.3 Main challenges of Requirements Engineering

The overall process of Requirements Engineering must overcome several challenges, among which end-user involvement, information extraction and requirements validation.

Involving end-users

As already hinted, the necessity to actively involve end-users of a future IT system during its specification and development steps has long been advocated, most notably by the proponents of User-Centred Design, which is also known as participatory design [Schuler and Namioka, 1993] or contextual design [Beyer and Holtzblatt, 1998]. Indeed, involving end-users in the expression of their needs and in the definition of an adapted and viable solution helps to avoid resentment and resistance towards a new information system infrastructure, as well as to stimulate productivity [Vosburgh et al., 1984].

The comprehensive survey led by the management consulting services provider Robbins-Gioia in 2001 precisely pointed out that project failure is not only defined by objective criteria, but also by the perception of the respondents [Robbins-Gioia, 2002]. The subjectivity of the respondent appears as a result of whether the person took an active role in the project or not.

Besides, as a matter of fact, end-users know "how business is done" in the environment for which software is being developed. They know the qualities and the flaws of the information systems currently used, and therefore have the ability to state what could be done to improve it [Fischer, 2002; Illich, 1973].

Extracting information

The main challenges when extracting requirements from stakeholders are all about *who* should be involved, *what* can be considered relevant information and *how* can it be extracted.

Indeed, each class of stakeholders provides different classes of requirements, although the latter often overlap. The knowledge and vital information can be distributed among various sources, which can have contradictory visions and interpretations. Therefore, selecting representatives for each class of stakeholders is very important.

Once the panel of intervenors is selected, *articulation* problems may appear since the stakeholders providing information are usually experts in their application domain but not in the process of engineering software. Therefore, this may lead to:

- *confusion*: they may not know what they want or misunderstand the issues;
- *improper expectations*: they may know what they want but it is inappropriate or unrealistic; or they may even not be aware of their real needs;
- *difficult articulation*: they may be aware of their needs but unable to express them in a coherent form; they may unintentionally hold back critical information because it has become part of their tacit knowledge;
- *unclear articulation*: they may express their needs with ease, but still face shortcomings of the natural language, such as ambiguity, inaccuracy or inconsistency [Wilson et al., 1997]; nevertheless, a general statement of objectives is not sufficient to start a relevant requirements analysis phase [Pressman, 2000];
- *inappropriate prioritisation*: they may struggle to identify precedences, and even be unwilling to prioritise and make trade-offs.

Beyond these issues lays the possible alteration of the information, typically because of *biases*. The latter can occur because of the previously mentioned resentment and resistance factors, which can lead to "lies" and/or "omissions". Moreover, the fact of being "studied" may also lead to a change of behaviour from the selected intervenors, which is known as the *Hawthorne effect* [Mayo, 1933].

The intrinsic nature of requirements is itself an important factor to manage. Indeed, during the Requirements Engineering process, the requirements are led to change and migrate as the users learn and grow. And since the extracted requirements are diverse and conflicting, they might be difficult to integrate and evaluate.

Validating requirements

As we have seen, the specification must be formal enough for the analysts and developers, but also understandable enough to be validated by the other stakeholders. Parallel *articulation* problems may appear, since the software engineers are experts in development and not in the users application domain. This problem is increased by the users and developers having different vocabulary, terms, and concepts. Bridging the communication gap between all parties is one of the most difficult problems of requirement engineering, but is absolutely indispensable in practice [Andriole, 1994].

2.1.4 How could end-users become major actors of Requirements Engineering?

It appears quite clearly that the requirements analysis phase is necessary to define the stakes of a software engineering project within an organisation. The main challenges lie in finding ways to involve actively the stakeholders (especially the end-users) and to help them express as clearly as possible their true and priority needs, while bridging the communication gaps to manage validation and agreement among all parties. Achieving this enables to produce a sound Software Requirements Specification that will be crucial for the subsequent conception and implementation phases.

2.2 Data Modelling

In this section, we investigate the particular case of *Data Modelling*, which plays a pivotal role in the realm of Requirements Engineering. Indeed, accurately eliciting and validating data user requirements is vital to build a reliable

documentation of the application domain, and therefore a reliable information system. *Database Engineering* precisely focuses on data modelling, where these requirements are typically expressed by means of a conceptual schema (or model), that is, an abstract view of the application domain. We therefore introduce basic notions of Database Engineering, as well as its specific challenges regarding requirements elicitation and validation.

2.2.1 Database Engineering

Defining the application domain of a software engineering project and structuring the information that needs to be manipulated by the system are the first steps of *Database Engineering*. This process of designing and implementing a database that has to meet specific user requirements has been described extensively in the literature [Batini et al., 1992] and has been available for several decades in CASE tools. It consists of four main subprocesses depicted in Fig. 2.1:

(a) *Conceptual design* which aims at expressing user requirements into a conceptual schema, that is, a technology-independent abstract specification of the future database. Such a schema is also known as a *PIM* (Platform-Independent Model) in the *UML* (Object Management Group) community;

(b) *Logical design*, which produces an operational logical schema (Platform-Specific Model or *PSM*) that translates the constructs of the conceptual schema according to a specific technology family without loss of semantics (e.g. Relational, XML, Object-Oriented). The transformational approach to Database Engineering [Hainaut, 2006] allows this process to be automated;

(c) *Physical design*, which augments the logical schema with performance-oriented constructs and parameters, such as indexes, buffer management strategies or lock management policies;

(d) *Coding*, which translates the physical schema (and some other artefacts) into the DDL (*Data Definition Language*) of the database management system.

The *transformational approach* [Hainaut, 2006] allows database engineers to automate the production of logical and physical schemas from their conceptual counterpart. Afterwards, from these schemas, well-mastered (semi) automated techniques, that have long been studied in the database research community and applied in industry, allow to produce the artefacts of the final application: interfaces, programs, database code, etc. [Schewe and Thalheim, 2005].

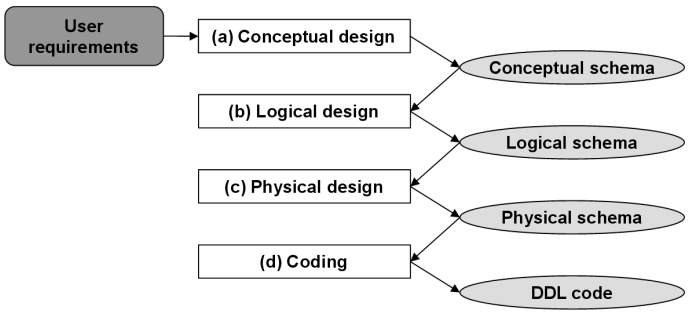


Figure 2.1: The Database Engineering process

The process leading to the production of the conceptual schema representing the requirements is therefore the most complex step of Database Engineering. We will hence focus on this process, that is *Conceptual design*, and more, we are interested in elaborating an approach allowing to reduce the potential gap between actual user requirements and their translation in the conceptual schema. Various techniques exist to elicit data requirements [Batini et al., 1992], such as the analysis of corporate documents and interviews of stakeholders. However, they usually do not actively and interactively involve end-users. Still, as we have seen in Section 2.1.3, the necessity to actively involve end-users of a future IT system during its specification and development steps has proven to be necessary.

Before going any further, let us mention that, in the Database community, the term *model* denotes what the UML community commonly calls a metamodel (e.g. the *relational model*). A database schema is hence an instance of a definite database model, that is, a UML model, represented by a class diagram. Among database models, the Entity-Relationship (ER) model has long been considered one of best mediums to express conceptual requirements [Shoval and Shiran, 1997]. 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, this statement has proved over-optimistic in many situations. It appears that the ER formalism, despite its merits, often fails to meet its objectives as an intuitive and reliable communication medium in which end-users are involved. The reason is easy to grasp: a conceptual schema is just a graphical presentation of a large and complex set of 1st and 2nd order predicates. Fig. 2.2 shows a small conceptual schema and its formal expression according to the GER formalism [Hainaut, 2006]. The intrinsic complexity of the requirements has been concealed by the apparent intuitiveness of the ER graphical

notation but it has not disappeared*. An in-depth comprehension of an ER schema implies the understanding of such non trivial concepts as sets, non-1st normal form relations, algebraic operators (projection, join, etc.), candidate keys and functional dependencies.

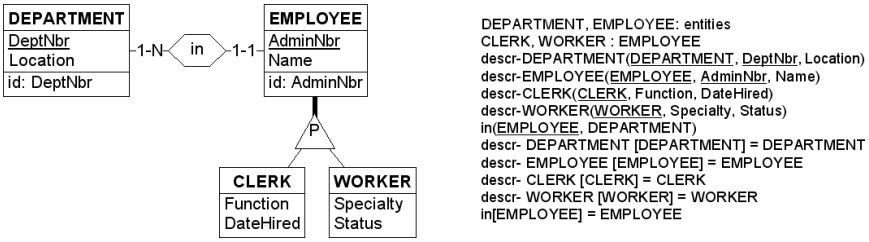


Figure 2.2: An ER schema and its formal expression

2.2.2 Database Reverse Engineering

Reverse engineering a piece of software consists, among other things, in recovering or reconstructing its functional specifications, starting mainly from the source code of the programs [Chikofsky and Cross, 1990; Hall, 1992; Hainaut, 2002]. When applied to databases, reverse engineering typically aims at recovering the database requirements (i.e. the conceptual schema) from multiple system artefacts that are usually obtained through schema transformation:

- documentation (when available);
- DDL code of the database;
- data instances;
- screens, reports and forms;
- source code of application programs.

Database reverse engineering traditionally consists of four main processes, which are illustrated in Fig. 2.3:

(a) *Data structure extraction*, which aims at extracting the raw physical schema of the database, including explicit and implicit constructs. A construct (structures or properties) is called explicit when it has been declared in the DDL code, while implicit constructs are implemented

*For instance, the example schema conveys the following statements: `DEPARTMENT` and `EMPLOYEE` are entity types; `CLERK` and `WORKER` are subtypes of `EMPLOYEE`, for which they form a partition; A `DEPARTMENT` can be described by `Location` and `DeptNumber`, the latter being an identifier for that entity type; An `EMPLOYEE` is `in` a `DEPARTMENT`; and so on.

through artefacts external to the database, such as code sections in application programs;

(b) *Refinement* enriches the raw physical schema with additional constructs and constraints elicited through the analysis of the application programs and other sources;

(c) *Cleaning* removes the physical constructs (such as indexes) for producing the logical schema;

(d) *Data structure conceptualisation*, in which a plausible conceptual schema is derived from the logical schema.

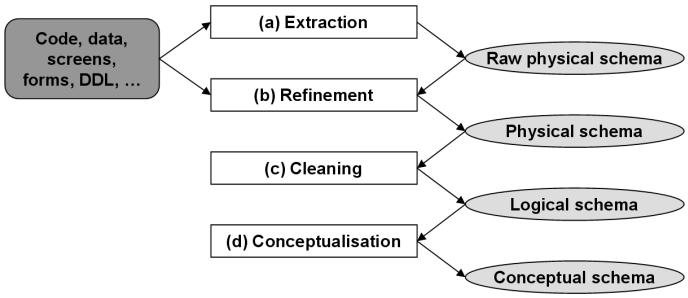


Figure 2.3: The database reverse engineering process

Database Reverse Engineering mainly focus on legacy systems, but is extensible to other problems. One can therefore already sense the opportunity of applying its principles on artefacts that are well-defined and under our control. Among those artefacts, the most meaningful for end-users is undoubtedly the user interface.

2.2.3 The Generic Entity-Relationship model

For the purpose of our research, let us introduce the Generic Entity-Relationship (GER) model [Hainaut, 1989], which is an extended Entity-Relationship model including, among others, the concepts of *schema*, *entity type*, *domain*, *attribute*, *relationship type*, *key*, as well as various constraints. The GER model encompasses the three main levels of abstractions for database schemas, namely *conceptual*, *logical* and *physical*, and serves as a generic pivot model between the major database paradigms including ER, relational, object-oriented, object-relational, files structures, network, hierarchical and XML. Let us now present and illustrate this model according to its different levels of abstraction.

Conceptual schemas

First of all, a conceptual schema mainly specifies *entity types*, *relationship types* and *attributes*. Entity types represent the main concepts of the application domain, which can be organised into *IS-A* hierarchies defining supertypes and subtypes. Such hierarchies can be total and/or disjoint. Total (T) means that a supertype must be specialised in at least one subtype. Disjoint (D) means that a supertype can be specialised in at most one subtype. A partition (noted P) corresponds to an IS-A hierarchy that is both total and disjoint.

Relationship types represent relationships between entity types. A relationship type has two or more roles. A role has a cardinality constraint [i-j], that specifies in how many relationships an entity can appear with this role. A relationship type with exactly two roles is called *binary*, while a relationship type with more than two roles is generally called *n-ary*.

Entity types and relationship types can have attributes, which can be either *atomic* (a.k.a. *simple*) or *compound*. A compound attribute is an attribute that is made of at least one sub-level attribute (simple or compound).

Attributes are also characterised by a cardinality constraint [i-j] specifying how many values can be associated with a parent instance. The *minimum cardinality* i states how many attribute values *must* be associated, while the *maximum cardinality* j corresponds to maximum number of values that *can* be associated, knowing that $0 \leq i \leq j$.

A *mandatory* (respectively *optional*) attribute is an attribute for which the minimum cardinality is equal 1 (respectively 0). A *single-valued* (respectively *multivalued*) attribute is an attribute for which the maximum cardinality is 1 (respectively >1). By default, the cardinality constraint of an attribute is [1-1].

Entity types and relationship types may also be given possibly complex constraints, which are expressed through the concept of *group*. A group is a logical set of elements (attributes, roles and/or other groups) attached to a parent object (entity type, relationship type or compound attribute). The most common types of constraints that can be defined on a group include:

- *primary identifier* (`id`) : the elements of the group form the main identifier of the parent object. A parent object can have at most one primary identifier. All components of an `id` group must be mandatory;
- *secondary identifier* (`id'`) : the elements of the group make up a secondary identifier of the parent object. A parent object can have several secondary identifiers;
- *coexistence* (`coex`): either all elements of the group have a value or none for any instance of the parent object;

- *exclusive* (`excl`): among the elements of the group, at most one can have a value for any instance of the parent object;
- *at-least-one* (`at-lst-1`): Among the elements of the group, at least one must have a value for any instance of the parent object;
- *exactly-one* (`exact-1`) Among the elements of the group, one and only one can have a value for any instance of the parent object. This corresponds to the combination of the *exclusive* and *at-least-one* constraints.

Figure 2.4 depicts an example of GER conceptual schema. We can observe that this schema includes entity types `PERSON`, `CUSTOMER`, `SUPPLIER`, `ORDER` and `PRODUCT`. The entity type `PERSON` has two disjoint subtypes, namely `CUSTOMER` and `SUPPLIER`.

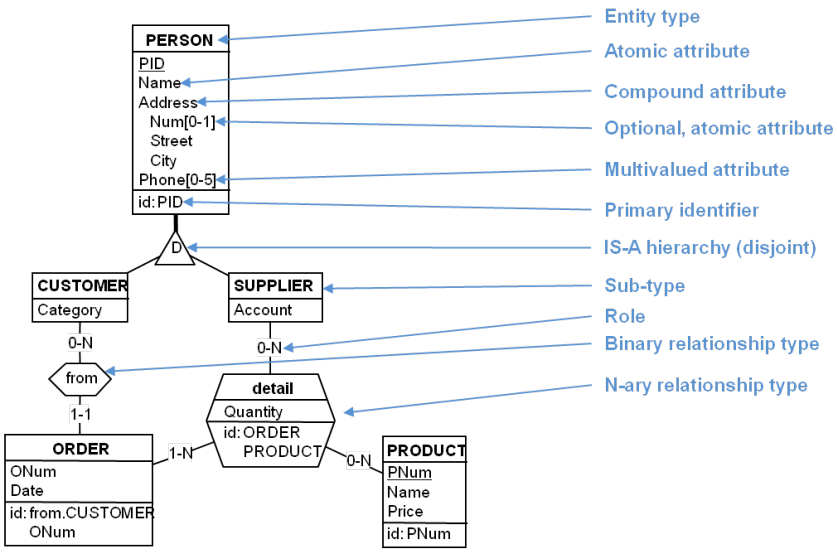


Figure 2.4: Sample GER conceptual schema.

The relationship type `from` is binary while `detail` is ternary. Besides, there cannot exist more than one `detail` relationship with the same `ORDER` and `PRODUCT` entities. Each `ORDER` entity appears in exactly one `from` relationship (cardinality [1-1]), and in at least one `detail` relationship (cardinality [1-N]).

For the entity type `PERSON`, the attribute `Name` is atomic, single-valued and mandatory. Among the components of the compound attribute `Address`, the `Num` is atomic, single-valued and optional (cardinality [0-1]). `Phone` is multivalued and optional, with 0 to 5 values per entity.

Finally, `PID` is the identifier of `PERSON`, while the identifier of `ORDER` is made of external entity type `from.CUSTOMER` and of local attribute `ONum`.

In the scope of this thesis, we will mainly focus on this level of abstraction.

Logical schemas

The second level of abstraction concerns logical schemas, which are platform-dependent data structure definitions, that must comply with a given data model. The most commonly used families of models include the relational model, the network model (CODASYL DBTG), the hierarchical model (IMS), the standard file model (COBOL, C, RPG, BASIC), the shallow model (TOTAL, IMAGE), the object-oriented model and the object-relational model (SQL3), as well as models expressed though XML schemas.

A logical schema basically uses the same schema constructs as the ones presented in Section 2.2.3 for conceptual schemas, but depending on the logical model the same schema constructs are called differently. For instance, a GER entity type (respectively attribute) is called a *table* (respectively *column*) in the relational terminology, and *record type* (respectively *field*) in a CODASYL schema.

Each logical model has its own set of *allowed* schema constructs. For instance, a relational schema may not comprise relationship types, compound attributes, multivalued attributes and *IS-A* hierarchies. Such illegal constructs must be expressed by equivalent constructs of the target logical model, when such alternatives are available. It can happen that some fragments of a conceptual schema cannot be fully translated into the logical schema.

In addition to the ones described above, new schema constructs may also appear at the logical level, including:

- *Referential constraint* (`ref`) : An inter-group constraint between an *origin group* (`ref` group) and a *target group*, stating that each instance of the origin group must correspond to an instance of the target group. The target group must represent an identifier (`id` or `id'`). A referential constraint is called a *foreign key* in the relational model.
- *Inclusion constraint* (`incl`) : An inter-group constraint where each instance of the origin group must be an instance of the target group. Here, the target group does not need to be an identifier (generalisation of the referential constraint).
- *Equality constraint* (`equ`) : A referential constraint between an origin group r and a target group i, combined with an inclusion constraint defined from the i to r.

- *Typed multivalued attribute*: In a conceptual schema, multivalued attributes represent sets of values, i.e. unstructured collections of distinct values. At the logical level, we can distinguish six possible implementations of such attributes:
  - *Set*: unstructured collection of distinct elements (default);
  - *Bag*: unstructured collection of (not necessarily distinct) elements;
  - *Unique list*: sequenced collection of distinct elements;
  - *List*: sequenced collection of (not necessarily distinct) elements;
  - *Unique array*: indexed sequence of cells that can each contain an element. The elements are distinct;
  - *Array*: indexed sequence of cells that can each contain an element.

An example fragment of logical schema is given in Figure 2.5. This relational schema corresponds to an approximate translation of the conceptual schema depicted in Figure 2.2.3, based on seven tables. Table `PERSON` has mandatory columns (`PID`, `NAME`, `ADD_STREET` and `ADD_CITY`) and one optional (nullable) column, `ADD_NUM`. Its primary identifier is {`PID`}. Column {`PID`} of `ORDER` is a foreign key to `CUSTOMER` (targeting its primary id). The group {`PID`, `ONUM`} of `DETAIL` is a multicomponent foreign key. In addition, there is an inclusion constraint from {`PID`, `ONUM`} of `ORDER` to {`PID`, `ONUM`} of `DETAIL`. Combining these two constraints translates into an equality constraint (`equ`). {`PID`} of `CUSTOMER` is both a primary id and a foreign key to `PERSON`.

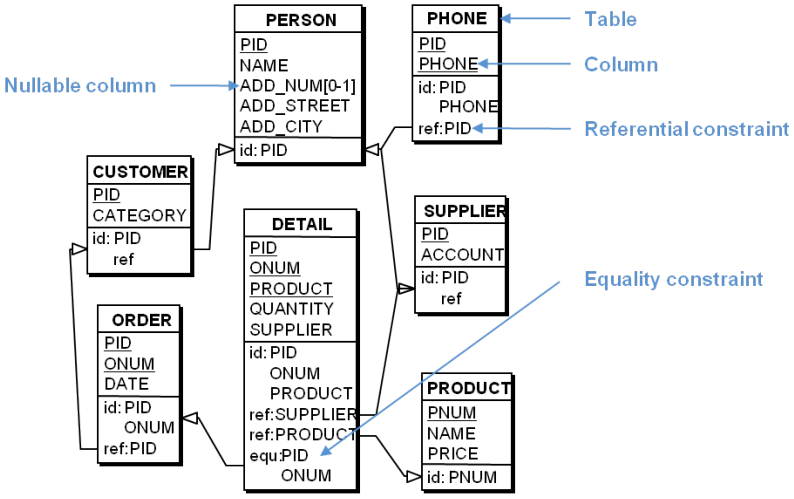


Figure 2.5: Sample GER logical schema, approximate relational translation of the conceptual schema of Figure 2.4.

Physical schemas

Finally, a physical schema is a logical schema enriched with all the information needed to implement efficiently the database on top of a given data management system. This includes DMS-dependent technical specifications such as indexes, physical device and site assignment, page size, file size, buffer management or access right policies. Due to their large variety, it is not easy to propose a general model covering all possible physical constructs, but we should at least mention the two following concepts:

- *record collection*, which is an abstraction of file, data set, tablespace, dbspace and any record repository in which data is permanently stored;
- *access key* (`acc`), which represents any path providing a fast and selective access to records that satisfy a definite criterion; indexes, indexed set (DBTG), access path, hash files, inverted files, indexed sequential organisations all are concrete instances of the concept of access key.

Figure 2.6 depicts a physical GER schema that derives from the logical schema of Figure 2.5. This schema is made up of seven tables and three collections. Collection `PERS_FILE` stores instances of tables `PERSON`, `CUSTOMER`, `SUPPLIER` and `PHONE`. The primary identifiers and some foreign keys are supported by an access key (groups denoted by `acc`). Access keys are also associated with two regular columns (`PERSON.NAME` and `PRODUCT.NAME`).

2.2.4 The transformational approach

Any process that consists in deriving artefacts from other artefacts relies on such techniques as renaming, translating, restructuring, replacing, refining and abstracting, which basically are *transformations*. Most Database Engineering processes can be therefore formalised as chains of elementary schema and data transformations that preserve some of their aspects, such as its information contents [Hainaut, 2006].

Schema transformation

Roughly speaking, an elementary schema transformation consists in deriving a target schema S' from a source schema S by replacing construct C (possibly empty) in S with a new construct C' (possibly empty). C (respectively C') is empty when the transformation consists in adding (respectively removing) a construct. Adding an attribute to an entity type, replacing a relationship type by an equivalent entity type or by a foreign key and replacing an attribute by an entity type (Figure 2.8) are some examples of schema transformations.

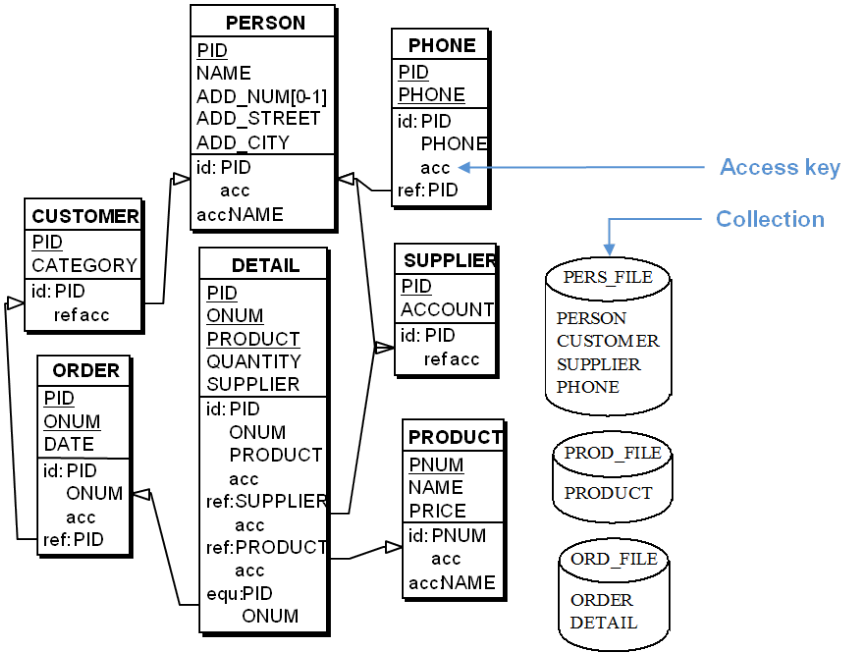


Figure 2.6: Sample fragment of a GER physical schema.

More formally, we can define a transformation as follows:

Definition 2.1. A *transformation* Σ is a couple of mappings $\langle T, t\rangle$ such that, $C' = T(C)$ and $c' = t(c)$, where c is any instance of C and c' the corresponding instance of C'. ⌟

Figure 2.7 illustrates how schema transformation can be defined as a couple of structural and instance mappings. A *structural mapping* T is a rewriting rule that specifies how to modify the schema while an *instance mapping* t states how to compute the instance set of C' from the instances of C.

There are several ways to express a structural mapping T. For example, T can be defined (1) as a couple of predicates defining the minimal source precondition and the maximal target postcondition, (2) as a couple of source and target patterns or (3) through a procedure made up of removing, adding, and renaming operators acting on elementary schema objects. Mapping t will be specified by an algebraic formula, a calculus expression or even through an explicit procedure.

Any transformation Σ can be given an inverse transformation $\Sigma' = \langle T', t'\rangle$, such that $T'(T(C)) = C$. Furthermore, a transformation Σ and its inverse

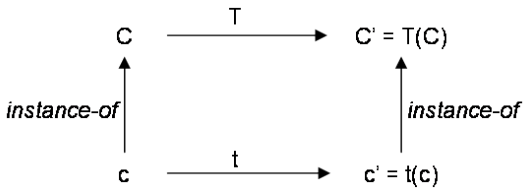


Figure 2.7: Schema transformation defined as a couple of structural and instance mappings.

transformation Σ') are called *semantics-preserving* if they verify $t'(t(c)) = c$.

Figure 2.8 shows a popular way to convert an attribute into an entity type (structural mapping T), and back (structural mapping T'). The instance mapping, that is not shown, would describe how each instance of source attribute `A2` is converted into an entity type `EA2` and a relationship type `R`. Let us notice that the concept of semantics (or information contents) preservation is more complex, but this definition is sufficient in this context. A more comprehensive definition can be found in [Hainaut, 2006].

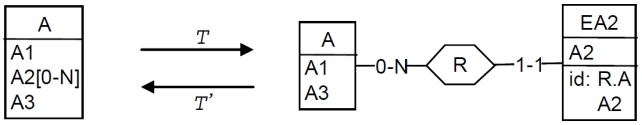


Figure 2.8: Pattern-based representation of the structural mapping of *ATTRIBUTE-to-ET* transformation that replaces a multivalued attribute (`A2`) by an entity type (`EA2`) and a relationship type (`R`).

Practically, the application of a transformation will be specified by its signature, that identifies the source objects and provides the names of the new target objects. For example, the signatures of the transformations of Figure 2.8 are:

$$\begin{array}{lcll} T & : & (\texttt{EA2}, \texttt{R}) & \leftarrow \mathit{ATTRIBUTE\text{-}to\text{-}ET}(\texttt{A}, \texttt{A2}) \\ T' & : & (\texttt{A2}) & \leftarrow \mathit{ET\text{-}to\text{-}ATTRIBUTE}(\texttt{EA2}) \end{array}$$

Transformations such as those in Figure 2.8 include names (`A`, `A1`, `R`, `EA2`, etc.) that actually are variable names. Substituting names of objects of an actual schema for these abstract names provides fully or partially instantiated transformations. For example:

$$(\texttt{'PHONE'}, \texttt{'has'}) \leftarrow \mathit{ATTRIBUTE\text{-}to\text{-}ET}(\texttt{'CUSTOMER'},\texttt{' Phone'})$$

specifies the transformation of attribute `Phone` of entity type `CUSTOMER`. Similarly,

$$(\texttt{EA2}, \texttt{R}) \leftarrow \mathit{ATTRIBUTE\text{-}to\text{-}ET}(\texttt{'CUSTOMER'},\texttt{A2})$$

specifies the family of transformations of any attribute of `CUSTOMER` entity type.

The concept of transformation is valid whatever the granularity of the object it applies to. For instance, transforming a conceptual schema CS into an equivalent physical schema PS can be modelled as a (complex) semantics-preserving transformation $\mathit{CS\text{-}to\text{-}PS} = \langle \mathit{CS\text{-}to\text{-}PS}, \mathit{cs\text{-}to\text{-}ps} \rangle$ in such a way that $PS = \mathit{CS\text{-}to\text{-}PS}(CS)$. This transformation has an inverse, $\mathit{PS\text{-}to\text{-}CS} = \langle \mathit{PS\text{-}to\text{-}CS}, \mathit{ps\text{-}to\text{-}cs} \rangle$, so that $CS = \mathit{PS\text{-}to\text{-}CS}(PS)$.

Compound schema transformation

A compound transformation $\Sigma = \Sigma_2 \circ \Sigma_1$ is obtained by applying Σ_2 on the database (schema and data) that results from the application of Σ_1 [Hainaut, 1996]. Most complex Database Engineering processes, particularly database design and reverse engineering, can be modelled as compound semantics-preserving transformations. For instance, transformation $\mathit{CS\text{-}to\text{-}PS}$ referred to here above actually is a compound transformation, since it comprises logical design, that transforms a conceptual schema into a logical schema, followed by physical design, that transforms the logical schema into a physical schema [Batini et al., 1992]. So, the database design process can be modelled by transformation $\mathit{CS\text{-}to\text{-}PS} = \mathit{LS\text{-}to\text{-}PS} \circ \mathit{CS\text{-}to\text{-}LS}$, while the reverse engineering process is modelled by $\mathit{PS\text{-}to\text{-}CS} = \mathit{LS\text{-}to\text{-}CS} \circ \mathit{PS\text{-}to\text{-}LS}$.

Transformation history and schema mapping

The *history* of an engineering process is the formal trace of the transformations that were carried out during its execution. Each transformation is entirely specified by its signature, while the sequence of these signatures reflects the order in which the transformations were carried out. The history of a process provides the basis for such operations as undoing and replaying parts of the process, which guarantees the traceability of the source and target artefacts.

In particular, it formally and completely defines the mapping between a source schema and its target counterpart when the latter was produced by means of a transformational process. Indeed, the chain of transformations that originates from any definite source object precisely designates the resulting objects in the target schema, as well as the way they were produced. However, the history approach to mapping specification has proved complex, mostly for the

three following reasons [Hainaut et al., 1996]. First, a history includes information that is useless for schema migration, especially when considering that the signatures often include additional information for undoing and inverting transformations. Second, making histories evolve consistently over time is far from trivial. Finally, the exploratory nature of engineering processes causes "real" histories not to be linear.

Therefore, simpler mappings are often preferred, even though they are less powerful. For instance, [Hick and Hainaut, 2006] proposed the use of a lightweight technique based on stamp propagation. Each source object receives a unique stamp that is propagated to all objects resulting from the successive transformations. When comparing the source and target schemas, the objects that have the same stamp exhibit a pattern that uniquely identifies the transformation that was applied on the source object. This approach is valid provided that (1) only a limited set of transformations is used and (2) the transformation chain from each source object is short (one or two operations). Fortunately, these conditions are almost always met in real database design.

2.2.5 How could conceptual analysis become accessible to end-users?

In this discussion, we focus on the production of the conceptual schema representing the requirements, which is the most complex step of Database Engineering. As we have seen, various techniques exist to elicit data requirements, but do not actively and interactively involve end-users. The GER model, which is a powerful mean of formalising requirements (most notably thanks to the transformational approach), suffers from its lack of expressiveness for the laymen. The question that therefore arises is how could we take advantage of that model and data modelling techniques to elicit data requirements, while bridging the gap with stakeholders that are not computer specialists?

2.3 Prototyping

2.3.1 A bridge between two worlds

Prototyping (also know as *Rapid Prototyping* (RP) or *User Interface Rapid Prototyping* (UIRP)) is a well-known software engineering technique [Gomaa, 1983; Lantz, 1986] that was introduced to deal with the main problems in the popularly used sequential approach to software development, especially the fact that errors and problems in the requirements definition frequently did not emerge until after the final product was used by the client.

Based on [Connell and Shafer, 1995] and [Pomberger et al., 1991], we can therefore define a prototype as a dynamic and interactive visual working model of user requirements, which should be easily modifiable and extensible while not necessarily being representative of the complete system. It is rather a communication tool for developers, customers and future end-users by providing the latter with a physical representation of key parts of the system before implementation.

This artefact can be used for non exclusive purposes, namely requirements validation (*exploratory prototyping*), design validation (*experimental prototyping*) or incremental software development (*evolutionary prototyping*). According to its purpose, the prototype may range from low (paper sketches for instance) to high (such as functional applications) fidelity, can be more or less detailed, and be conceived to be reused or not. Anyhow, as explained by [Schneider, 1996], such an artefact carries a tremendously valuable knowledge that should be documented and maintained throughout any software engineering process.

One of the major assets of prototyping lies in the fact that from the user's perspective, forms, and more specifically electronic forms, are more natural and intuitive than usual text-based descriptions, paper models and conceptual formalisms to express data requirements [Choobineh et al., 1992], while making the semantics of the underlying data understandable [Terwilliger et al., 2006].

The comprehension of user interfaces has improved in organisations thanks to the increasing use and training level in the field of information technologies. As explained by Illich, by providing the users with convivial tools, we allow them "to invest the world with their meaning, to enrich the environment with the fruits of their vision and to use them for the accomplishment of a purpose they have chosen" [Illich, 1973]. Indeed, convivial tools encourage users to be active and generate themselves extensions to the artefacts given to them, which can potentially break down the barriers between consumers and designers [Brown and Duguid, 2000].

Moreover, as observed by [Fischer, 2002], many computer users and designers today are henceforth domain professionals, competent practitioners, and discretionary users. They should not be considered naive or incompetent users: they worry about tasks, they are motivated to contribute and to create quality products, they care about personal growth, and they want to have convivial tools that make them independent of tools created by an "elite" of designers and to which the non designer users (that is, the majority of computer users) must subjugate.

In parallel, a form contains data structures that can be seen as the physical implementation of a particular view of the conceptual schema of the database,

since the transition from forms to a semantic model has been shown to be tractable [Rollinson and Roberts, 1998]. Consequently, a prototype can be analysed (for instance using database reverse engineering techniques) to recover requirements such as its underlying conceptual schema [Hainaut, 2002]. In this spirit, Ravid and Berry suggested a general method for fine-tuning any rapid prototyping method [Ravid and Berry, 2000]. They identified the categories of requirements information that a prototype user interface may contain. One of these categories concerns *"the application's data model, data dictionary, and data-processing capabilities"*, which is our main concern.

In conclusion, prototyping is a joint design activity characterised as a meeting between two languages, that of the developer and that of the user's work world.

2.3.2 Prototyping as the expression of formal requirements

Let us now demonstrate that the form-based interfaces of a prototype can indeed express the requirements of a formal schema. Fig. 2.9 (left) shows an electronic form of a complex data structure derived from the conceptual schema of Fig. 2.2. Its information contents is represented at the right side as a hierarchical record type, each field of which represents either an elementary form field or a possibly multivalued grouping of such fields. From the theoretical point of view, there is no formal guarantee that both schemas are equivalent, that is, that the record type of Fig. 2.9 provides the same information as the schema of Fig. 2.2. Though they are not *exactly* equivalent, we can show that the record type captures most of the information of the conceptual schema.

For this, we will use a demonstration approach based on transformational techniques. According to the latter, every engineering process can be modelled as a chain of schema transformations [Hainaut, 2006]. A transformation operator is defined by a rewriting rule that substitutes a target schema construct for a source construct. The most interesting operators are semantics-preserving, in that the source and target constructs convey the same semantics (they have the same meaning though presented differently). Fig. 2.10 illustrates two important semantics-preserving operators, namely *attribute to entity type mutation* and *upward inheritance*. 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 there are sufficient considering the scope of this study.

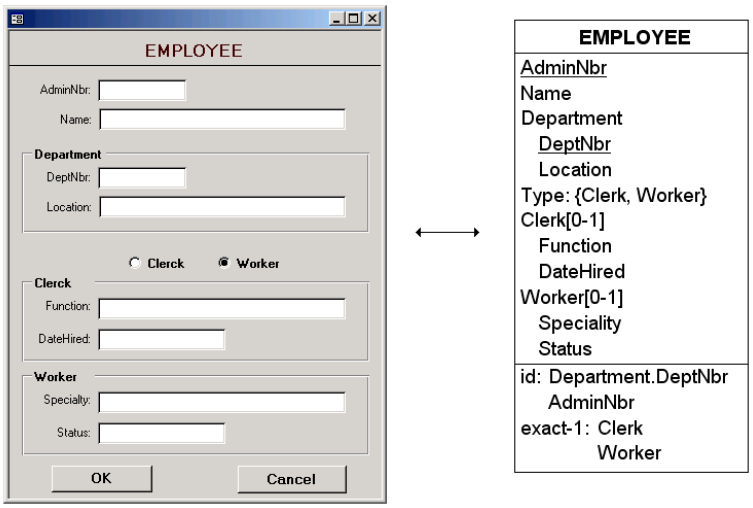


Figure 2.9: An electronic form and its information contents.

Now, we can apply transformation T1 on entity type `DEPARTMENT` of Fig. 2.2, which yields the compound attribute `Department` in Fig. 2.9. Then, we apply transformation T2 on the subtypes `CLERK` and `WORKER` of the schema of Fig. 2.2. They are transformed into attributes `Clerk`, `Worker` and `Type` of the schema of Fig. 2.9. Since the transformations are semantics-preserving, they can be applied in the reverse way, in such a way that the schema of Fig. 2.9 can be transformed in that of Fig. 2.2.

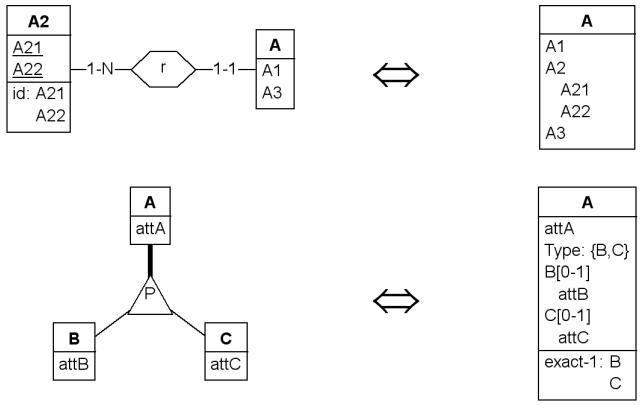


Figure 2.10: T1 (top) and T2 (bottom), two (almost) semantics-preserving transformations.

As a conclusion, we can consider that the electronic form of Fig. 2.9 expresses in an intuitive way the information requirements formally expressed in Fig. 2.2. Let us now see how this principle can be put in motion.

2.3.3 How could Prototyping be used as a two-way communication channel?

Prototyping has proven to be an efficient technique to elicit and validate requirements. In particular, form-based interfaces appear to be a powerful means of communication between all the stakeholders of an Software Engineering project, since they can be used to express formal data requirements. Moreover, reverse engineering techniques can be applied to a form-based prototype to derive valuable data user requirements. However, prototypes are still mainly used as a one-way communication channel, since they are designed by analysts rather than the end-users. From this observation, we can wonder if it could not be possible to make prototyping accessible to any of the stakeholders, in order to let them transparently express formal requirements on which could be applied transformational techniques.

Chapter 3

State of the art and Research questions

Whereas creating form-based user interfaces from existing databases is a well mastered and common design process, several research projects have also been focusing on applying reverse engineering techniques to form-based artefacts. In particular, such approaches, applied to legacy form-based mediums in order to recover existing database schemas, are presented in Section 3.1, while similar approaches, applied to prototypical form-based mediums in order to define new database schemas, are presented in Section 3.2.

Examining these approaches offers a certain number of perspectives, which are presented in Section 3.3. In particular, overcoming the limitations of the existing approaches implies managing several key problems inherent to the different disciplines that would need to interoperate in order to perform an interactive conceptual analysis based on the reverse engineering of prototypical user-drawn form-based interfaces.

We notably address the main activities of Database Forward Engineering in Section 3.4, the use of Database Reverse Engineering to extract data models from form-based interfaces in Section 3.5, the use of Prototyping to express and validate requirements in Section 3.6, and User-involvement in Section 3.7.

3.1 Reverse Engineering of Legacy Form-Based Artefacts

Back in 1984, Batini et al. studied paper forms as a widely used mean to collect and communicate data in the office environment. Since forms are a very

natural type of user requirements and an effective starting point in data base design of office application, they developed a methodology to integrate them into a non redundant semantic data model [Batini et al., 1984]. The method relies on 4 manual steps. During the *form analysis*, each form is decomposed into areas of related fields which are recorded in a glossary. The *area design* stage then extracts an EER schema for each area of a form. Finally, the *form design* and *interschema integration* phases deal with the integration of each schema into a single application schema. The possible integration conflicts must by solved using analyst expertise, which makes this approach unsuitable for regular end-users.

Mfourga presented a framework for extracting an entity-relationship schema from a set of form model schemas of an operational relational database [Mfourga, 1997]. The first core activity consists in acquiring the set of forms (structures and instances) of the legacy database to define form model schemas that gathers structural information and constraints among data. A static analysis is then performed to elicit the structural components and their interrelationships based on the logical and physical composition of the forms. Subsequently, a dynamic analysis leads to discover constraints as well as functional and existence dependencies. The process of extracting Entity-Relationship schemas then relies on the form model schemas as input, and goes through six steps, namely entity derivation, relationship derivation, attribute attachment, cardinality determination, conceptual normalisation and incremental binary schema integration. The overall process requires user interaction.

In order to understand the information and process logic embodied in a given legacy system, Stroulia et al. developed the twofold CELLEST method for reverse engineering its interfaces [Stroulia et al., 1999]. First, a map of the system interface is built based on the traces of interaction and navigation between end-users and the system. During this *interface mapping*, snapshots of the system screens are taken and their associated keystrokes are recorded, before being analysed and clustered to build an *interface graph*. Secondly, an abstract model of the user's task is constructed using the interface map and task-specific traces during the *task and domain modelling*. The exchanged information is classified based on two orthogonal dimensions: their type of triggering *action* (tell or ask) and their *scope* (system constants, user variables, task constants or problem variables). Based on this classification, a screen transition diagram and the associated flow of information are specified for each task. This task model is subsequently used to generate graphical interfaces that can be used as a reference to improve the legacy interfaces associated with the elicited tasks.

Heeseok and Cheonsoo explored the links between form-based user inter-

faces and the conceptual schema of the application domain from a reverse engineering perspective, and developed a method to extract the semantic of legacy applications from forms [Lee and Yoo, 2000]. They developed the form driven object-oriented reverse engineering method (FORE), which uses electronic screen forms as original input source and assumes that a form process consists of a task using a single form. The method consists of five different phases. (1) The form usage analysis captures the form structure and the user interaction within the legacy application, using an agent program to store them into a form knowledge store. (2) The form object slicing slices the form knowledge store into semantic units based on the input structure. (3) The object structure modelling creates a structure model from of the objects and their structural relations, which are identified from the previous semantic units. (4) The scenario design elaborates an object process action scenario diagram, based on target scenarios of the business processes. (5) The model integration integrates the structure models into a single one and resolves the structural conflicts, based on common objects and the collaboration of objects operation.

Astrova and Stantic developed an approach to migrate the Deep Web to the Semantic Web, using reverse engineering of relational databases to ontologies [Astrova and Stantic, 2005]. Their approach uses HTML pages as the main input and goes through three basic steps: (1) Form Model Schema extraction, which consists in analysing the HTML pages (analysis of the structure, analysis of the data, integration of the individual schemas) to extract a form model schema; (2) Schema transformation, where the form model schema is transformed into an ontology formulated in Frame-Logic using mapping rules; (3) Data migration, whose goal is to create ontological instances from data contained in the pages using table understanding techniques, in order to form a knowledge base whose schema is defined by the ontology.

3.2 Reverse Engineering of Form-Based Prototypes

3.2.1 Existing approaches

These principles have also been used on form-based prototypes, in order to specify new databases schemas. Let us review the main contributions in this area.

Choobineh et al. explored a form-based approach for database analysis and design [Choobineh et al., 1992]. Their initial standing point was that end-users could communicate many requirements through the forms they use, thanks to familiarity. Moreover, the most widely used data are gathered or reported in forms. Indeed, as they stated, a form is a structured collection of variables

(form fields) that are appropriately formatted for data entry and display. A form type defines the structure, constraints, and presentation of the form fields. Static properties of form fields include their type (primitive or user-defined), presentation (template), structure, origin (provided by the user or the system, computed from form fields, transferred from another form, depending on another field in the current form, depending on fields in other forms) and constraints (optionality, default values, value domain). From these observations, they developed the Form Definition System and the Expert Database Design System. The Form Definition System (FDS) provides an editor to create forms. The form layout component enables to enter form fields caption and example values. The interface component provided various interface widgets. The command component provides input/output functions and form properties. The inference component makes inferences on the grouping of form fields, dependencies among the form fields, etc., to generate positive and negative examples that end-users could validate or reject. Finally, the form abstraction base stores all the created forms. The FDS turned out to be most useful in providing a common vocabulary and goals among end-users and data processing professionals, rather than in providing exhaustive requirements collection by end-users. The Expert Database Design System incrementally produces an Entity-Relationship Diagram (ERD) based on the successive analysis of a set of forms (typically obtained from the form abstraction base). The process relies upon the following phases. (1) The form selection phase automatically selects the next form to analyse, based on fields linked from one form to another. (2) The entity identification phase identifies the possible entities within a form, using identifiers, dependencies, grouped fields, etc. (3) Once the entities are identified, the attribute attachment phase identifies their attributes using functional dependencies between fields of a single form. (4) The relation identification phase then creates the connections between the entities using functional dependencies between fields of different forms. (5) Afterwards, the cardinality identification phase makes decisions on the cardinalities of entities in a relationship. (6) Finally, the consistency phase ensures the consistency and integrity of the schema diagram throughout its incremental construction, by checking properties such as the completeness and uniqueness of the concepts, the preservation of the existing mappings between form fields, etc. To assure completeness of the whole database design, the authors suggested the approach to be combined into a methodology using other sources, such as natural language descriptions.

Kösters et al. introduced a requirements analysis method called *FLUID* which combines user interface and domain requirements analysis [Kösters et al., 1996]. The first stage of the method develops an initial domain model (comprising classes, attributes and relationships) and a task model which describes

the activities the user can accomplish with the help of the system. During a second step, the task model serves as a basis for the completion of the domain model as well as for the development of a user interface model. The initial domain model is developed through standard elicitation techniques.

Rollinson and Roberts studied the problem of non-expert customisation of database user interfaces in [Rollinson and Roberts, 1998]. To do so, they examined graphical form interfaces to identify which and how controls were used. They then examined how they were combined to represent information, and how they were mapped to an underlying data model. Finally, they identified and classified the conflicts that may occur among graphical form interfaces. They also described a hypothetical form modification system that would allow end-users to create and modify forms. The system would use a drag-and-drop Graphical User Interface (GUI) builder to create the form-based interfaces. An information extraction component would analyse the information conveyed by the interface and produce a semantic data model. A comparison component would compare the semantic data model of each interface. And finally, a storage component would hold copies of editable interfaces. This form modification system would however be restricted as it would not be fill-in, and would not allow changes implying a modification of the underlying database system. They developed the Form Interface Schema (FIS) as a mean to represent user interfaces, which consists of a directed graph where each node represents an instance of an Abstract Interaction Object with different properties (name, label, constraints and so on). From this Form Interface Schema, they developed a set of graph-oriented transformations to extract an Extended Entity-Relationship schema describing an interface's information content, based on graph rewriting rules as well as a classification of interschema relationships and conflicts.

More recently, Terwilliger et al. defined the formal GUAVA (GUi As View) framework to use the user interface directly as a conceptual model, by exploiting the hierarchical nature of forms-based user interfaces to provide a simple representation of their informational content, including the relationships between forms [Terwilliger et al., 2006]. First, the complete structure of the user interface is automatically represented in a hierarchical structure called a GUAVA-tree (g-tree) based on the user interface controls. Then the g-trees are translated into simple relational table structures with a natural schema against which querying is simple. Finally, a database designer can transform the natural schema into the underlying physical database schema using a collection of database operators.

Regarding web-based applications, Rode et al. investigated the feasibility of end-user web engineering for webmasters without programming experience [Rode et al., 2005]. Since most of the existing CASE/RAD tools are designed for

experienced developers, they developed *Click*, a prototypical tool for the end-user development of web application involving non professional programmers. The end-users develop their application by building pages from scratch or using predefined templates, then placing components (static layout elements, input and output elements, ...) or directly defining the database structure. The underlying layout code (HTML representation) and behaviour code (high-level functions implemented on top of PHP, such as send mail, save to database, go to page) are updated on the fly so that the application can be previewed instantly, following a *design-at-runtime* paradigm. A to-do list warns the end-user of undesirable or incoherent states (inexpressive default labels, missing links or pages, ...). Click provides several layers of programming support ranging from the mere customisation of existing templates to the edition of the PHP code itself, making it accessible for different levels of designing and programming knowledge.

Yang et al. also inquired about the WYSIWYG development of Data Driven Web Applications [Yang et al., 2008]. They developed the *AppForge* system to helps end-users to graphically specify the components of a form-based web application (such as a Yahoo! Group application), while transparently generating the underlying application model (specification of page views, application logic and database schema) on the fly. The users are provided with a GUI that allows them to create applications, user roles, pages (which contains forms and views), forms (data input), views (data view and update) and containers. Behind the GUI, the back-end system consists of an Application Creation System which maintains the application model based on the developers action , and an Application Runtime System which puts the model in action and connects it to a relational database. While being primarily destined to profane end-users, AppForge proved to be challenging for less experienced user, in particular because of the multiple levels of abstraction and parametrisation that they had to switch between while developing their application (creators, user, specific user, ...), which may suggest that regular end-users are not made to be complex application developers.

3.2.2 Core principles

The main lessons from this review are that reverse engineering form-based prototypes is a well-known problem, and the existing approaches all rely on the same core principles, which can be illustrated by Fig. 3.1:

- build a set of form-based interfaces;
- extract the underlying form model;
- translate the form model into a working data model;

- progressively build an integrated data model by looking for structural redundancies as well as constraints and dependencies.

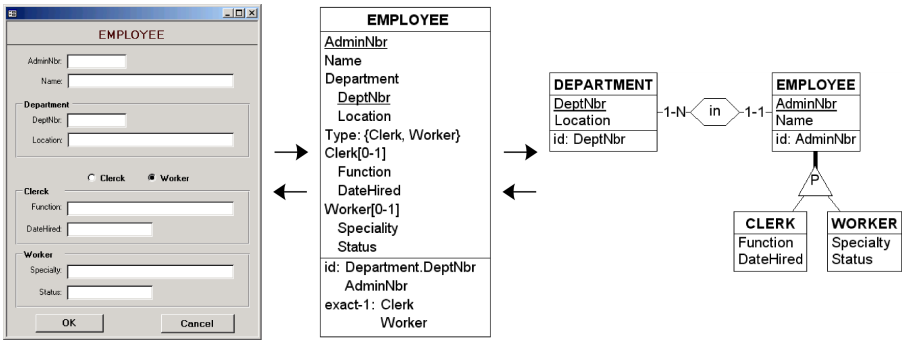


Figure 3.1: Illustration of the core principles for reverse engineering form-based interfaces: each interface is transformed into a "raw" data model, then it is progressively refined and integrated.

3.2.3 Limitations

However, we observe that the number of studies on the subject is limited (especially recently), and several limitations must be underlined in most of these approaches:

- the end-users are not involved intimately in the overall process;
- 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, ...) and ontological ambiguity (polysemy, homography, synonymy);
- the output schema often lacks refinement, such as hierarchies, existence constraints or functional dependencies;
- the use of examples (either through static statements or dynamic interaction) is not systematically used to elicit constraints and dependencies (when the latter are available);
- the underlying form model of the interfaces must often be constructed by analysing the physical composition (layout) before the informational composition (content) of the form;
- the tools provided for the drawing of the interfaces are either not dedicated to this purpose (e.g. Xfig), or de facto destined to professional designers or analysts rather than end-users;

- the prototypical form-based interfaces do not use a generic language that would enable GUI generation of an application on any target platform;
- the final version of the integrated data model is not systematically submitted to the end-users for a final validation;
- the possible evolution of the data model is not considered.

3.3 Research perspectives

Overcoming these limitations implies managing several key problems inherent to the different disciplines that would need to interoperate in order to perform an interactive conceptual analysis based on the reverse engineering of prototypical user-drawn form-based interfaces. In particular, we need to focus on managing and unifying the terminology and structure of the intended data model, its enrichment to include hierarchies, constraints and dependencies, as well as the generation of applicative components. Since we also want to involve intimately the end-users, we must also provide them with adequate means to express requirements through prototyping, and map these requirements to their database engineering counterparts.

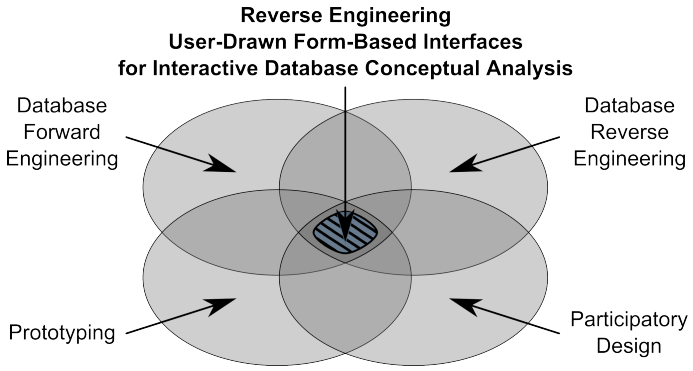


Figure 3.2: The overlay of different disciplines to perform an interactive conceptual analysis based on the reverse engineering of prototypical user-drawn form-based interfaces.

As suggested by Fig. 3.2, these challenges are dealt by specific disciplines, but their concerns and subsequent processing overlay in the context of our inquiry. Regarding Database forward engineering, we notably need to clarify terminological and structural ambiguities, elicit constraints and dependencies, handle schema integration and generate applicative components. For our purpose, Database reverse engineering mainly addresses the extraction of data

models from form-based interfaces. As for Prototyping, we must allow users to express concepts and requirements through form-based interfaces, then validate them through a playable lightweight application. Finally, since we want to emphasise user-involvement, we need to find ways to involve them and possibly tailor and integrate existing techniques. For each of these issues, let us now present the problem and its context, as well as existing methods to tackle it.

3.4 Encompassing Database forward engineering main activites

Among the various activities of Database forward engineering, we notably need to clarify terminological and structural ambiguities, elicit constraints and dependencies, handle schema integration and generate applicative components.

3.4.1 Clarifying terminological and structural ambiguities

One of the main challenges of realising a sound conceptual schema concerns the standardisation of the terminology and structures used to express similar and different concepts, so that elements representing the same notion bear the same name and elements representing different notions bear different names. The first step of this process consists in identifying and mapping elements that may correspond semantically to each other, which is known as *schema matching* [Rahm and Bernstein, 2001]. How can we therefore *investigate* a collection of existing heterogeneous schemas to find and solve possible ambiguities among their elements?

Terminological ambiguity

The first type of problems is the *terminological ambiguity* (also known as *syntactic heterogeneity*) between elements, which can occur because elements *seem* to bear similar names. Naming variabilities mostly result from the richness of written natural language (paronymy, polysemy, homography, synonymy, gender, singular and plural forms, ...) and possible spelling mistakes. Besides, a schema can be designed by possibly multiple analysts, based on possibly multiple sources of information using a non consistent vocabulary, which potentially increases this phenomenon. In order to standardise the terminology used in a given schema, the first step therefore consists in discovering and grouping elements that are similar either by their *spelling* or their *meaning*, which can precisely cause ambiguity. Note that though we here focus on the naming of schemas elements, these observations also apply to other properties such as the cardinalities, value types and value sizes of attributes.

First of all, the *orthographic similarity* between two elements relies on the spelling of these elements names, that is, the strings of characters composing them. Identifying orthographically similar strings is a well-known problem, usually dealt by using *String Metrics* [Cohen et al., 2003]. Such metrics calculates a similarity or dissimilarity score between two strings for matching and/or comparison. Other kinds of metrics can also be used, for instance based on the phonetic distance. For this purpose, one can mention *Soundex*, which is a phonetic algorithm for indexing names by sound [Jacobs, 1982].

Among the wide variety of reliable string metrics, we can mention Levenshtein's [Navarro, 2001] and Jaro-Winkler's metrics [Winkler, 1990], which are the most popular for dealing with word comparison. Levenshtein's metrics is defined as the minimum number of edits needed to transform a source string into a target string, based on the number of necessary insertion, deletion, or substitution of a single character. The greater the Levenshtein distance, the more different the strings are. Jaro-Winkler's metrics is based on the number of matching characters and transpositions between a source string into a target string, adjusted by the comparison of the initial characters of both strings. The distance ranges from 0 (different) to 1 (equal). For instance, the Levenshtein's and Jaro-Winkler's metrics yield the results showed in Table 3.1 when comparing the strings "Name", "First Name", "Last Name" and "Family Name'. Note that the metrics were used through different existing Java libraries that are presented in Section 11.3.1 *.

String 1	String 2	Levenshtein	Jaro Winkler
Name	Name	0	1.0
Name	First Name	6	0.0*
Name	Last Name	5	0.45
Name	Family Name	7	0.56
First Name	Last Name	3	0.83
First Name	Family Name	5	0.80
Last Name	Family Name	5	0.75

Table 3.1: Levenshtein's and Jaro-Winkler's distance applied to example strings.

Secondly, the *ontological similarity* between two elements relies on the meaning of these elements names. An *ontology* is a formal specification of the conceptualisation of a given knowledge, usually within a given domain [Gruber, 1995]. It defines a vocabulary that explicitly expresses the concepts of that domain, their classification and properties, as well as the relationships between

*Strangely enough, all these reference libraries implementing the Jaro Winkler metrics yield the unexpected result 0.0 when comparing "Name" and "First Name".

them. Ontologies are used in various fields and applications of software engineering, such as artificial intelligence, semantic web, biomedical informatics and so on. Several languages have been developed to support the encoding of ontologies. One of the most prominent ones is *Web Ontology Language* (OWL) and its family of languages, which are endorsed by the World Wide Web Consortium. Using such languages, a wide variety of domain-specific ontologies has been developed, most notably for scientific (biology, medicine, computer science, ...) and business fields.

Ontologies can therefore be useful to track down similarities of meaning among a set of words. Consider for instance the strings "Primary provider" and "Alternative supplier". They are not orthographically close (Levenshtein finds them similar at 16 and Jaro-Winkler at 0.52), but one may notice the nearness of meaning of the words "provider" and "supplier". Besides, the adjectives "primary" and "alternative" suggest that a classification may exist between these possible synonyms. Thesaurus and Dictionaries can also be useful to track down similarities of meaning among a set of words, and may also target specific domains, e.g. UMLS for the medical field [Hersh et al., 2000].

To clarify the terminological ambiguities between the elements of a given schema, it is therefore possible to combine String Metrics, Ontologies, Thesaurus and Dictionaries in order to compare these elements. Elements that are orthographically and/or ontologically similar would therefore be considered *semantically similar*, and would need to be arbitrated with the help of domain experts.

Structural redundancy

The second type of similarity that may occur is the *structural redundancy*. Typically, we can observe that attribute owners (such as entity types, relationship types and compound attributes) can share attributes bearing the same names. Take for instance the schema of Fig. 3.3 : the entity type `Person`, the entity type `Reservation` and the compound attribute `Manager` all have attributes named `First Name` and `Last Name`. Note that although we here focus on the naming of schema elements, these observations also apply to other properties such as cardinalities, value types and value sizes of attributes.

The observation of entity types sharing attribute with the same name suggest that these elements may represent different degrees of similarity. Let us consider two entity types $E1$ and $E2$ sharing two attributes A and B (Fig. 3.4(a)), and review the most common cases of structural redundancies:

- *equality*: The two entity types represent the same concept, but were assigned different names, for instance because of one of the reasons men-

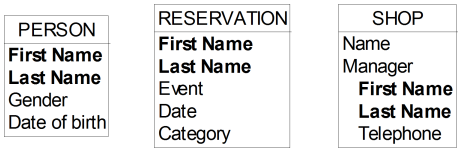


Figure 3.3: A simple schema with structural redundancies

tioned about terminological ambiguities (recall for instance "Provider" and "Supplier"). Such entity types should be merged into a single concept (Fig. 3.4(b));

- *union*: The two entity types partially represent the same concept, and could be seen as specialising a higher concept non explicitly expressed (Fig. 3.4(c)). For instance, within a same company, a `Clerk` and a `Sales Representative` are specialisations of the concept of `Employee`;
- *specialisation*: One of the two elements is a specialisation of the other (Fig. 3.4(d)). For instance, one could argue that a `Sales Representative` is an enhanced `Shop Assistant`;
- *complementarity*: One of the two entity types actually refers to the other (Fig. 3.4(e)). For instance, `Reservation` refers to `Person` in Fig. 3.3;
- *difference*: Finally, two entity types can also fortuitously share a same set of attributes, while being intrinsically different. For instance, a `Subcontractor` and a `Supplier` may share properties such as `Name` and `Address`, but still represent different concepts.

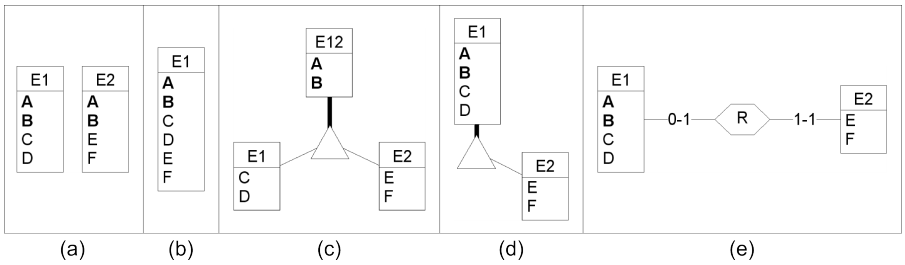


Figure 3.4: Typical cases of structural redundancies.

Besides, two entity types can also be considered structurally redundant if one of them contains components that can be obtain by *derivation* from components of the other one. This is often the case in form-based interfaces, when the content of a field is automatically computed from the values of fields coming from other forms. In order to standardise the terminology of the attributes

owners of a given schema, the first step therefore consists in discovering and grouping elements that share structural redundancies.

Given the tree-like structure of the GER model, the problem of mining structural redundancies is actually alike the problem of tree mining [Chi et al., 2005], and more precisely *frequent embedded subtrees mining in rooted unordered trees* [Jiménez et al., 2008]. A *rooted unordered tree* is a particular connected, directed and acyclic graph $G = (V, E)$ comprising a set V of vertices (or nodes) together with a set E of edges (or lines). Such a tree has a node (called *root*) from which it is possible to reach all the other vertices (*descendants*) in the tree, and does not provide any order among siblings. Each other node has one and only one parent node. Given such a tree, a *bottom-up subtree* is obtained by taking one vertex from G and all its descendants. An *induced subtree* is a bottom-up subtree from which leaf nodes (i.e. nodes with no descendant) have been repeatedly removed, while an *embedded subtree* is another particular bottom-up subtree from which nodes have been removed without breaking the ancestor relationship between the vertices of G.

In this context, entity types can be seen as root nodes, compound attributes as intermediary nodes, simple attributes and roles as leaves, and the order of the attributes is precisely irrelevant, as we explored in [Vilz et al., 2006]. Several well known tree mining algorithms have been developed to tackle this issue, such as Zaki's SLEUTH [Zaki, 2005], Asai et al.'s UNOT [Asai et al., 2003], Termier et al.'s TreeFinder [Termier et al., 2002] or Chehreghani et al.'s TDU [Chehreghani et al., 2007]. Tree based approaches are suitable for complex and deep graphs, however we observe that the structure of user-drawn interfaces is usually quite simple (the path from the root to the deepest node in the tree rarely uses more than a few edges), if only for legibility and usability. Indeed, Choobineh et al. noticed that "most forms have a shallow (i.e. few levels) and narrow (few nodes per level) structure because of human information processing limitations" [Choobineh et al., 1992]. This might imply that simpler algorithms may be more appropriate.

For the purpose of discovering structural redundancies, *Formal concept analysis* (FCA) is also a popular approach [Wille, 2005]. It relies on the definition of a *formal context* $\mathbb{K} = (G, M, I)$, where G is a set of formal objects, M a set of formal attributes, and I a binary relationship defining which attributes of M can be associated to which objects of G. Consequently, a *formal concept* of $\mathbb{K}$ is a pair (A, B) with $A \subseteq G$ and $B \subseteq M$, so that the set of all attributes shared by the objects of A is identical with B, and on the other hand A is also the set of all objects that have all attributes in B.

Thus, FCA can therefore be seen as a conceptual clustering technique providing descriptions and hierarchisation for the abstract concepts or data units

it produces. In this context and for a given schema, entity types, relationship types and compound attributes can be seen as formal objects, while attributes and roles can be seen as formal attributes. Searching the different formal concepts contained in the schema, and hierarchically grouping them according to their shared attributes could consequently highlight structural redundancies in a given schema.

To clarify the structural ambiguities between the attribute owners of a given schema, it is therefore possible to use tree mining and/or FCA in order to compare these elements. Elements that share embedded subtrees would therefore be considered *structurally similar*, and would need to be arbitrated with the help of domain experts.

3.4.2 Eliciting implicit constraints and dependencies

When conceiving a conceptual schema, it is important to define a set of predicates that will guarantee that once the subsequent database is implemented and operational, any changes 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. Among the different *constraints* usually considered to obtain such a reliable database, let us recall the most common ones:

- *domains of values*, which may 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, ...;
- *cardinality constraints*, which define the minimal (typically zero or one) and maximal numbers (typically one or infinite) of occurrence(s) of given attributes and roles;
- *existence constraints*, which define how optional components (attributes and roles) may influence each other. For two components A and B, these constraints may be:
  - *coexistence*, which implies that A and B must always be not null simultaneously;
  - *at-most-one*, 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;

- *identifiers*, which are a set of components which, taken collectively, allow to identify uniquely the given instance of a given entity type;
- *functional dependencies*, which express constraints between sets of attributes;

The challenge here is more about uncovering possible undetected constraints than expressing them directly. Indeed, while traditional database elicitation techniques usually may yield most of the relevant constraints during the design of the conceptual schema, analysing the content of the subsequent database (or at least, a set of relevant data samples) may highlight constraints that remained unmentioned, maybe because the domain experts were not aware of them, or (more probably) because they are part of some tacit knowledge. How can we therefore *nurture* an existing schema to elicit and express constraints and dependencies based on data samples?

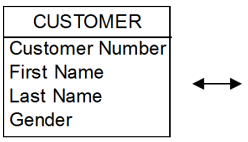
The relational model of a database

Since we need to consider our set of schemas not as a "rigid skeleton", but as a "living being" (i.e. an operational and populated database), 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, 1970].

In the relational model, all the data is represented through *relation*s (also know as *tables*). A relation is structured using *attribute*s (a.k.a. *fields* or *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 with respect to its domain. Intuitively, relations and attributes of the relational model correspond to entity types and attributes in the GER model, as illustrated in Figure 3.5*.

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. Let us note $t[A]$ the restriction of a tuple t to the set of attributes A (called *projection* of t onto A). For instance, from the tuples visible in Figure 3.5, we could assume that `Customer Number` and `Last Name` form an identifier for the relation `Customer`, since there are no tuples having the same combination of values. More formally, we therefore have:

*Most of these customers are fictional characters from the acclaimed TV show Battlestar Galactica.



	CUSTOMER			
tuples	**Customer Number**	**First Name**	**Last Name**	**Gender**
t1	23	Bill	Adama	Male
t2	25	Lee	Adama	Male
t3	35	Bill	Dog	Male
t4	46	Caprica	Six	Female
t5	52	Kara	Thrace	Female
t6	53	Sharon	Valerii	Female
...	...	...	...	...

Figure 3.5: The representation of a Customer using the GER and relational model.

Definition 3.1. Given a relation R, its associated sets of attributes $\mathcal{A} = \{a_1, ..., a_n\}$ and possible tuples $\mathcal{T} = \{t_1, ..., t_m\}$, $\bar{\mathcal{A}} \subseteq \mathcal{A}$ is *identifier* for R iff: $\nexists i, j \in [1, m] : t_i[\bar{\mathcal{A}}] = t_j[\bar{\mathcal{A}}] \wedge i \neq j$

An identifier is *minimal* if none of its members can be removed without jeopardising the identifying property. For instance, from the tuples visible in Figure 3.5, we could assume that `Customer Number` and `Last Name` is not a minimal identifier for the relation `Customer`, since we can remove `Last Name` and still identify each tuple using uniquely `Customer Number`.

In a database, each relation contains an implicit identifier that can is obtained by construction. However, it is preferable to define an explicit identifier for each relation. Since there can be several explicit identifiers, it is common to choose a minimal identifier as the sole *primary identifier*. Identifiers other than the primary one are said to be *secondary*. When it is impossible to define a primary identifier, a virtual attribute is added to the relation to play the role of *technical identifier*.

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 from a database. Given a relation R, a set of attributes $X \in R$ is said to functionally determine another set of attributes $Y \in R$, if and only if through all the possible tuples extending R, each value of X is associated with precisely one value of Y. This functional dependency is written $R : X \rightarrow Y$, with X called the *determinant* set of attributes and Y the *dependent* set of attributes. For instance, from the tuples visible in Figure 3.5, it seems that the functional dependency `Customer:First Name` $\rightarrow$ `Last Name` does not stand, 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` $\rightarrow$ `Gender` could be legit, but would need to be validated.

Armstrong's axioms are a set of inference rules used to infer all the functional dependencies on a relational database [Armstrong, 1974]. The axioms

are sound in that they generate only functional dependencies in the closure of a set of functional dependencies (denoted as F^+) when applied to that set (denoted as F). They are also complete in that repeated application of these rules will generate all functional dependencies in the closure F^+.

- Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$
- Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
- Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

In addition to these rules, the following derivate rules are also taken in account.

- Union: If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$
- Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- Pseudo Transitivity: If $A \rightarrow B$ and $CB \rightarrow D$ then $AC \rightarrow D$

For instance, If $A \rightarrow B$ is an FD, then, $(A, C \rightarrow B)$ is considered redundant.

Mining constraints and functional dependencies

Analysing the content of a database or a subset of data samples can intuitively lead to make possible assumptions on, e.g., the domains of values, the cardinalities of the attributes, their existence constraints and possibly their identifiers. 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 $t_i[A]$ have different values, this could suggest that A is in fact a primary or secondary identifier. The same kind of *induction* could be led on optional attributes to assess their possible existence constraints. However, mining functional dependencies is far less trivial.

Back in 1995, Ram presented four categories of heuristics to derive functional dependencies from an existing conceptual ER schema [Ram, 1995]. 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 heuristics 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 itself. This is a well-know 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 types of approach, which are so different that it is difficult to compare them qualitatively [Lopes et al., 2002].

The *candidate generate-and-test* approach is based on the partitioning of the database combined with a levelwise exploration and pruning of the search space, as in Huhtala et al.'s TANE [Huhtala et al., 1999], Novelli and Cicchetti's FUN [Novelli and Cicchetti, 2001], or Yao and Hamilton's FD_Mine [Yao and Hamilton, 2008]. The modus operandi of these algorithms is sensibly the same, and they mainly differ by the pruning rules that they use. First of all, the stripped partition database $\hat{r}$ of the relation r is calculated by partitioning the tuples into maximal equivalence classes based on their values. Then, the dependencies are computed level by level, starting with the singleton sets of attributes of r, and continuing with the combinations of non discarded attributes to test the dependencies of the form $X\backslash\{A\} \rightarrow A$ with $A \in X$ for each set X of the level.

The *minimal cover* approach relies on searching 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. [Lopes et al., 2000] and FastFDs, proposed by Wyss et al. [Wyss et al., 2001]. They also rely on the extraction of a stripped partition database from the initial relation, before computing agree sets of tuples. Maximal sets are then generated, which allows to define a minimum FD cover according to these maximal sets, based on hypergraph theory. DepMiner uses a levelwise search, while FastFDs uses a depth-first, heuristic-driven (DFHD) search.

Finally, *Formal concept analysis* (FCA) has also been used recently to find and represent logical implications in datasets [Priss, 2005], 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 [Baixeries, 2004], while Rancz et al. optimise an existing method introduced by [Correia, 2002], 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 [Rancz and Varga, 2008]. The latter authors also developed the subsequent FCAFuncDepMine software to detect functional dependencies in rela-

tional database tables [Rancz et al., 2008]. 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 [Flory, 1982].

To elicit the implicit possible constraints and dependencies of a given schema, it is therefore possible to use induction, dependency discovery algorithms and FCA on data samples. Such constraints evidently need to be arbitrated with the help of domain experts. However, existing approaches rely on massive pre-existing data sets, which could be problematic.

3.4.3 Handling schema integration

View integration

The ultimate objective of the Conceptual Design is to produce a single, integrated conceptual schema describing the future database. The final step of this database design process is therefore to handle *view integration*, which is a typical case of *database schema integration* [Batini et al., 1986]. View integration may be required because the stakeholders have different perspectives on the modelling of the information, which can induce different representations of the same concepts (different names, different constructs, different properties, etc.) that may turn out to be incompatible. It is therefore crucial to resolve these discrepancies and integrate the involved elements in order to obtain a sound and consistent conceptual schema.

View integration consists in identifying and comparing elements to integrate, unifying them before merging and restructuring them. The elements to consider are those involved in issues such as naming conflicts (typically homonyms and synonyms) and structural conflicts (such as construct mismatches, identifiers conflicts, cardinality conflicts, constraints conflicts, ...). The difficulty actually lies in conflict identification rather than in conflict resolution. Managing this process can therefore be simplified if we take in account the elements that we discussed for the previously mentioned challenges. How can we therefore *bind* elements of an existing schema?

Transformational techniques

First of all, as we just pointed out, the identification and subsequent unification of the elements depends on the problems previously discussed in Section 3.4.1. If the semantic and structural ambiguities are solved, we can here focus on merging and restructuring the necessary elements. Transformational techniques have proved to be particularly powerful to carry out this process and enable to

integrate similar objects into a unique, non-redundant structure, without any loss of semantics.

When dealing with multiple (sub)schemas, three main strategies naturally appear, as explained by Hainaut [Hainaut, 2009]:

- *n-ary* integration, which is suitable when several concepts appear in multiple (sub)schemas;
- *hierarchical* integration, which is suitable when the application domain is naturally decomposed into hierarchical subsystems;
- *incremental* integration, which is suitable for large and numerous schemas who do not prevail upon each other.

Hierarchical and incremental integration rely on a progressive *binary* integration of schemas, where corresponding elements are pairwisely merged. For the sake of simplification, we consider that two elements correspond to the same concept if they are semantically and/or structurally similar. Typically, two corresponding entity types A and B should be replaced with a merged entity type C, for which each attribute and role would find at least one correspondent in A and/or B. Merging attributes and roles implies arbitrating their differing properties, including cardinalities, domain of values and value types, and so on. Integrity constraints must also be propagated.

However, there is not always a strict identity between two concepts, other integration techniques must then be used to resolve redundancies [Spaccapietra et al., 1992]. For instance, if two objects are not of the same type, such as an entity type and a compound attribute corresponding to the same concept, transformational techniques are used to set them in conformity before integration, by typically extracting the compound attribute as an entity type.

3.4.4 Generating applicative components

From the conceptual schema to the prototype

Once the integrated conceptual schema representing the application domain has been produced, and in the perspective of integrating prototyping for the purpose of database forward engineering, it might be interesting to generate the database and interfaces to which it could be connected. Handing a playable prototype to domain experts could serve indeed as a valuable validation of the data specifications, since it would allow them to test various aspects of their requirements. How can we therefore *objectify* an existing schema into applicative components such as a database?

Transformations and CASE tools

As exposed in Chapter 2.2, producing the conceptual schema representing the requirements is the most complex step of Database Engineering. Once it is elaborated, the database engineers can subsequently automate the production of the platform-specific logical schema and the performance-oriented physical schema from their conceptual counterpart, using dedicated transformations plans, i.e. specifically defined sequences of standard transformations. Afterwards, from these schemas, well-mastered (semi) automated techniques, that have long been studied in the database research community and applied in industry, allow the artefacts of the final application to be produced: interfaces, programs, database code, etc.

These transformations and techniques are usually handled by dedicated CASE tools that allow to customise the necessary sequence of transformations to meet specific needs and target technologies. For instance, *DB-MAIN* is a data-oriented modelling CASE-tool dedicated to Database Application Engineering [DB-MAIN, 2010]. It is designed to help developers and analysts in the development, reverse-engineering, reengineering, migration, integration, maintenance and evolution of data-centred applications, mainly based on the Entity-Relationship model. DB-MAIN includes meta-model components that allow the users to develop transformational scripts, new functions and methods, as well as a Java library to interact with its internal repository.

3.5 Using Database reverse engineering to extract data models from form-based interfaces

In the Chapter 3.4, we discussed the challenges surrounding the definition of the conceptual schema, and more particularly the aspects that should be managed on an ongoing schema. To begin these refining processes, we could start from a raw schema obtained from the transformation of a set of form-based interfaces into a set of data schemas, as suggested by the principles of Database reverse engineering that have been exposed in Section 2.2.2, and the applications of these principles which were presented in Section 3. How can we therefore *adapt* a set form-based interfaces into data models to which we could apply the previously discussed processes of Database forward engineering?

3.5.1 Static information based on layout and content

First of all, the appearance of form-based interfaces contains a lot of information regarding the underlying data models. Indeed, a set of forms can be considered

a set of derived views of the data. To extract each view from the forms, the approaches of Section 3 usually start by analysing the layout of the forms to detect the labels of each widget and how they are logically and hierarchically structured. This step is not trivial, since the labels may not systematically be visible, and the choice and placement of the objects and their label may not always be intuitive and systematic, as illustrated in Fig. 3.6. In this example, we can for instance observe that:

- some elements do not bear any label, such as the group boxes for the product and the payment, as well as the radio buttons for the title of the customer;
- the placement of the label is not always the same, as it is sometimes at the left of the widgets, and sometimes on the top;
- the placement of the radio buttons is not always the same either: they are on the same line for the title of the customer, but not for the balance of the payment;
- and so on...

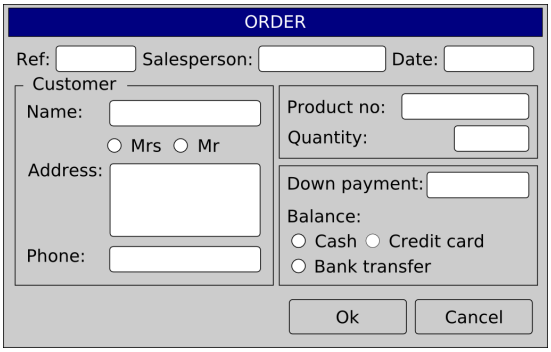


Figure 3.6: A form-based interface with unlabelled elements and unsystematic choice and placement of widgets.

Once the structure and labels of the forms are clarified, each given form F_i is translated into a data model M_i using injection-based *mapping rules* between elements of the chosen form model and data model, so that each widget of F_i has a deterministic counterpart in M_i. These mapping rules depend on the languages used to model and/or implement the source interfaces, as well as the target destination model. As it appears, the output data models will be more precise and detailed if it is possible to access the detailed specifications of the interfaces, which is not always possible with legacy interfaces.

3.5.2 Dynamic information

Besides, it is also possible to observe the execution of programmes to analyse the actions and inputs that must be processed, and how the programmes respond to these actions and inputs. If the database is accessible, one can also observe how the data evolves according to this execution. Investigating program behaviour using information gathered as the program is running is a well known problematic. For instance, *Program profiling* (also known as *Software profiling*) mainly focuses on determining which sections of a program could be optimised (typically in terms of overall speed and memory requirement), while *Program comprehension* focuses on acquiring knowledge about computer programs in order to facilitate reuse, inspection, maintenance, reverse engineering, reengineering, migration, and extension of existing software systems.

The data obtained typically through interpretation (for instance using the Virtual Machine in Java) or instrumentation of system's execution is used for such purposes as reverse engineering and debugging. Numerous dynamic analysis approaches have been proposed for this purpose, with a broad spectrum of different techniques and tools as a result, as presented by [Cornelissen et al., 2009]. According to the main objectives of the programme comprehension and the target programming platforms and languages, different methods can be combined, among which visualisation, program slicing, filtering, metrics, querying, ...

3.6 Prototyping to express and validate requirements

Among the various challenges of Prototyping, we notably need to allow end-users to express concepts through form-based interfaces, and conversely, to validate concepts through form-based interfaces.

3.6.1 Expressing requirements through form-based interfaces

How can users *represent* concepts and requirements by building themselves form-based interfaces?

Modelling form-based interfaces

In their most general definition, forms are a structured mean of displaying and collecting information for further processing. Originally, forms were materialised as paper documents sharing common parts and including blank fields to fill in the necessary information, typically for orders, requests, checks, ...

The introduction of electronic forms allowed for conveniently typing in the variable parts by providing a set of widgets accordingly to the type of expected data input (constrained or unconstrained, mono or multivalued, ...). These interactive forms, which have become a natural part of GUIs through a wide variety of applications and websites, usually use the following common *input* widgets:

- *textual input fields*, that allow input of a single or several lines of text;
- *radio buttons*, which usually allow to choose one value among several ones;
- *check boxes*, which usually allow to choose zero, one or several values among several ones;
- *push buttons* (to call a specific function or another screen, for instance to search a file or reset the whole form);
- *combo-boxes* (drop-down list that displays a list of items a user can select from).

On top of these simple widgets exist more specific or complex widgets, such as spin boxes (to adjust a value in an adjoining text box by either clicking on an up or down arrow), tree views (to display hierarchical information) or colour pickers.

Input widgets are usually combined with *output* (display) widgets, such as *labels* or *images*. Both categories of widgets can be structured using *group boxes* and *tables* within a given top level *window*.

Existing User Interface Description Languages

In the last decade, new classes of IT devices have emerged in conjunction with new interaction styles such as 3D interaction, virtual/mixed reality, tangible user interfaces, context-aware interfaces and recognition-based interfaces. Since there is a large variety of programming languages each offering its own model of GUI, there has been many research on *User Interface Description Languages* (UIDL).

Such languages enable designers to specify user interfaces using high-level constructs, and without worrying about implementation details. From the description of these abstract user interfaces, concrete user interfaces can then be (semi) automatically generated according to the chosen platforms and technologies. As described by [Luyten et al., 2004], the goals of UIDLs are to:

- capture the requirements for a user interface as an abstract definition that remains stable across a variety of platforms;

- enable the creation of a single user interface design for multiple devices and platforms;
- improve the reusability of a user interface;
- support evolution, extensibility and adaptability of a user interface;
- enable automated generation of user interface code.

Let us therefore briefly review a few recent UIDLs, which are mostly based on the eXtensible Markup Language (XML), which is a standard markup language that has notably become a standard recommendation of the *World Wide Web Consortium* (W3C) to model and carry structured data [W3C, 2010]. The tree-like structure of XML documents perfectly with the hierarchical structure of traditional form-based interfaces.

The *User Interface Markup Language* (UIML) is an XML-based language that allows the canonical description of user interfaces for different platforms [Ali et al., 2002]. A UIML document is structured in three different parts: a UI description, a peers section that defines mappings from the UIML document to external entities (target platform's rendering and application logic), and finally a template section that allows the reuse of already written elements. The UI is described as a set of interface interaction elements for which a presentation style (such as position, font style or colour), the content (text, images, etc.) and the possible user input events and resulting actions are specified. The interface is built using a rendered that interprets UIML on the client device (similar to the way a web browser renders an HTML file) or compiles it to another language (for instance HTML). However, the UI description is bound to the target language and device, which implies that a same interface may need separate UI descriptions if it has different targets.

The *eXtensible Interface Markup Language* (XIML) [Puerta and Eisenstein, 2002] is a representation language for interaction data that supports design, operation, organisation, and evaluation functions. It is able to relate the abstract and concrete data elements of an interface, while enabling knowledge-based systems to exploit the captured data. XIML is a hierarchically organised set of interface elements that are distributed into one or more of the different interface components. XIML predefines five basic interface components, namely task (the business process and user tasks), domain (the hierarchical set of all the objects and classes used), user (the hierarchical tree of the target end-users), dialog (the structured collection of elements that determine the possible interactions), and presentation (the hierarchy of concrete interaction objects used in the interface). The interaction data elements captured by the various XIML components can be linked together using relations, and be enriched by attributes (i.e. features or properties).

Teresa XML is the XML-compliant language that was developed inside the Teresa project, which is intended to be a transformation-based environment supporting the design and the generation of a concrete user interface for a specific type of platform [Paternò and Santoro, 2002]. The Teresa XML language is composed of a XML-description of the CTT notation [Mori et al., 2002] which was the first XML language for task models, as well as a language for describing user interfaces. Teresa XML specifies how the various *Abstract Interaction Objects* (AIO) composing the UI are organised, along with the specification of the UI dialog.

The *USer Interface eXtensible Markup Language* (UsiXML) is a XML-compliant markup language that describes the UI for multiple contexts of use such as Character User Interfaces (CUIs), Graphical User Interfaces, Auditory User Interfaces, and Multi-modal User Interfaces [Limbourg et al., 2004; Limbourg and Vanderdonckt, 2004]. This language allows interactive applications with different types of interaction techniques, modalities of use, and computing platforms to be described in a way that preserves the design independently from peculiar characteristics of physical computing platform. It was initiated by the exhaustive review of XML-compliant User Interface Description Languages led by [Souchon and Vanderdonckt, 2003].

Mapping UIDLs to Data Models

As we have seen in the Chapter 3.5, it is possible to define mapping rules between UIDLs and Data Models, which would transparently associate specific widgets to specific types of concepts. However, a major concern about these various UIDLs comes from the structure of the languages itself. Indeed, since they must express complex interfaces, layouts and behaviours, their XML structure becomes complex and difficult to read, which in turn may lead to very complex mappings to given data models.

Besides, such complex languages may scare laymen users that need to express simple concepts, because they would have to choose among too many widgets and would be challenged by the definition of appropriate layouts. These concerns may prevent them to focus on the content of the forms rather than the appearance. The users should therefore be given adequate tools to manage the drawing of accessible form-based interfaces.

3.6.2 Validating requirements through form-based interfaces

As we have seen in Chapter 2.3, prototypical interfaces can also be used to validate requirements. If applicative components can be generated from the

integrated conceptual schema that results of the analysis of the raw schema obtained from prototypical form-based interfaces, a prototypical application could be generated and testing as a final validation step. How can users therefore *wander* through a playable prototype to validate their requirements?

As we have seen in Section 3.4.4, the generation of applicative components can be relatively straightforward using transformations and CASE tools. The challenge here is to interconnect form-based interfaces with their underlying database and applicative components.

3.7 Managing User-Involvement

In the previous chapters, we presented challenges inherent to the disciplines that would need to interoperate in order to perform an interactive conceptual analysis based on the reverse engineering of prototypical user-drawn form-based interfaces. Still, one of our major concerns resides in the involvement of end-users, so that they can effectively and efficiently participate in the resolution of these challenges. How can we therefore handle user-involvement in this context?

3.7.1 Participatory Design Perspectives

[Bødker et al., 1993] address different recommendations regarding the application of Participatory Design. Among these recommendations, it appears that the design process should be situated within the users work but guided and arbitrated by the designers. It should encourage creativity and draw out tacit and shared knowledge, while simulating the future to aid in prediction and evaluation of design. Kensing and Blomberg also insist that end-users should participate in the analysis of needs and possibilities, the evaluation and selection of technology components, the design and prototyping of new technologies, organisational implementation, and ultimately, in decision making [Kensing and Blomberg, 1998].

In this context, the roles of the designer include coordination, facilitation, material preparation and managing the social interactions (which can be referred to as "Social Engineering"). Since it is not systematically possible for all those affected by the design effort to fully participate in the process, the choice of user participants, their responsibilities and accountabilities as well as their form of participation must be carefully considered and negotiated.

Besides, [Sanders, 2002] explains that to access the end-users experience and knowledge, the analysts can observe them under three perspectives: what they *say*, what they *do*, and what they *make*. Listening to people tells us what they

are able and willing to express in words (i.e., *explicit* knowledge) and watching people in their activities provides us with *observable* information, which may help us to understand their perceptions of experience. To grasp their *tacit* knowledge, we also need to understand how they feel in order to empathize with them. Observing end-users on the long run can also reveal unsuspected needs (*latent* knowledge).

Furthermore, Muller et al. drew a taxonomy of Participatory Design practices according to "who participates with whom in what" and at what stage of the development cycle this activity occurs in [Muller et al., 1993]. As it appears, we would like end-users to participate in the early database design activities, which would rather call for *co-development* settings.

In this context, [Grønbæk et al., 1997] precisely advocated users and designers to collectively explore the form, functionalities and context of applications through *cooperative prototyping*. Using adequate prototyping tools and the users actual work materials to allow case-based prototyping, they can apply their knowledge and experience as competent professionals in the design process. Besides, [Mogensen, 1992] and [Trigg et al., 1991] also noted that prototypes can act as "catalysts" and "triggers" for discussions, which may lead to mutual learning, since it provokes concrete experience.

3.7.2 Tailoring existing techniques

Arguably, using cooperative prototyping as a means to express, capture and validate data requirements implies tailoring and integrating the processes and existing techniques in order to suit the context of use, support the users skills and keep them focused and dedicated to the overall process.

Typically, since conceptual schemas are difficult to comprehend, we should find a way to transparently use form-based interfaces instead for the various steps of analysis and arbitration of the design process. The design and interaction with the prototype should be simple, intuitive and enjoyable, which could be threatened by the fact that existing UIDLs are complex and that we might require the acquisition of numerous data samples (which would be (too) demanding) for further analysis. Also, the available algorithms for mining structural dependencies and possible constraints are quite complex, which implies that their execution could be time-consuming and therefore interfere with the flow of the process. All these observations urge us to consider adapting or redefining the existing strategies to improve the experience of the end-users, so that their interactions with the data modelling process become more intuitive and transparent.

Part II

The RAINBOW Approach

In this part of the dissertation, we present the integrated RAINBOW approach to reverse engineer user-drawn form-based interfaces in order to perform an interactive database conceptual analysis. First, Chapter 4 introduces the approach and formalises its principles into a semi-automatic process. Each of its seven steps is subsequently detailed in a separate chapter and a proof-of-concept tool support is then presented. The principles and results of this proposal have notably been presented in international conferences [Ramdoyal et al., 2010, 2009; Vilz et al., 2006; Brogneaux et al., 2005a] and workshops [Ramdoyal, 2010; Ramdoyal et al., 2007; Brogneaux et al., 2005b].

Chapter 4

Proposal

4.1 Claim

As we have seen, providing a better requirements acquisition process for Database Engineering implies bridging the gap between end-users and analysts. Since the traditional ER schema has shown understandability limitations, this issue clearly calls for a better medium, which should be common to all the stakeholders and rich enough to convey relevant meaning and interactivity. For this purpose, we propose to **use user-drawn form-based interfaces as a two-way channel to express, capture and validate static data requirements with end-users by taking advantage of reverse engineering techniques**. More precisely, we claim that:

- Given:
  - An environment for which forms are a privileged way to exchange information;
  - Stakeholders familiar with form-based (computer) interaction and the application domain;
- We can:
  - Exploit the expressiveness of form-based user interfaces and prototypes;
  - Specialise and integrate standard techniques to help acquire and validate data specifications from existing artefacts;

- In order to:
  - Use form-based user interfaces as a two-way channel to communicate static data requirements between end-users and analysts;
  - And therefore transparently produce a conceptual schema of the application domain, including integrity constraints, existence constraints and functional dependencies.

Indeed, since existing artefacts can be used to recover the underlying requirements through well-mastered reverse engineering techniques, we advocate to use such tailored techniques in forward engineering by working with the virtual artefacts produced by the end-users (Fig. 4.1). This approach benefits from the advantages of rapid prototyping, while making the user a central actor of the process, and designing a set of simple semantic interfaces rather than a complete application.

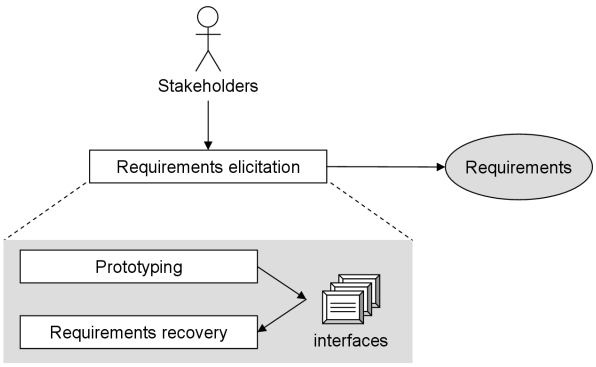


Figure 4.1: Using reverse engineering in a forward engineering perspective.

4.2 Context of use

The RAINBOW approach targets a certain type of context, which can be characterised by the following aspects:

- *Application type*: the modelled software engineering projects should be related to form-based data intensive applications; this is typically the case for business applications that need to frequently encode, share and access specific data;
- *Target companies*: the expected enterprises should be small to medium sized, such as a self-employed entrepreneur or a local chain of stores;

- *End-users profile*: the participating end-users should be at least familiar with form-based human-computer interactions, typically through office suites or form-based websites. Data modelling experience is not required, so that a regular white-collar or secretary would be an archetype of the expected kind of end-user. Still, the approach should also be accessible for to other types of users, such as database engineers, analysts or developers.

4.3 Founding principles

In order to formalise this approach, we need to take in account several specificities, among which:

- a high level of interaction with the end-users;
- the possibility to involve different levels of participants, ranging from laymen to experts, through a modular process;
- the need for a tool support accessible to end-users and useful to the analysts;
- the necessity to tailor existing techniques.

We indeed want to provide end-users with adequate tools to draw and specify by themselves the interfaces describing the underlying key concepts of their application domain, without having to worry about any application logic. Provided a little training and as previously explained, involving end-users in such processes may have a very positive impact. This is especially true in the Requirements Engineering process, for which it is essential to avoid mismatches between the actual needs of end-users and the way they are formalised.

By allowing end-users to build themselves a "light" prototype of the future application (in terms of command screens and information exchanges), we avoid the development of a "heavy" prototype, which is obviously an expensive task (it includes the development of a limited but operational application, whose components cannot usually be reused in the implementation of the final system) and reduces the costly presence of computer designers during the specification phase. In this context, the computer analysts rather appear as guides, whose roles are oriented towards the validation of requirements and the generation of complex code.

These principles are at the foundation of broader approaches, such as the *ReQuest* framework [Vilz et al., 2006], which provides a complete methodology and a set of tools to deal with the analysis, development and maintenance of web-based data-intensive applications. Regarding data modelling, that approach consists of four main steps: (a) inviting the end-users to draw the interfaces of the future application, (b) extracting of data structures from each

interface fragment into a logical model, (c) analysing the logical models to identify and resolve redundancies, (d) integrating and conceptualising the logical models. During the whole process, traceability is ensured, so that the conceptual structure corresponding to an interface component can be retrieved and conversely. The ReQuest framework also deals with dynamic aspects of the future application (such as task analysis, behaviour of the application, ...), while providing generators for several components of the future application (database, framework skeleton, ...), as illustrated in Fig.4.2.

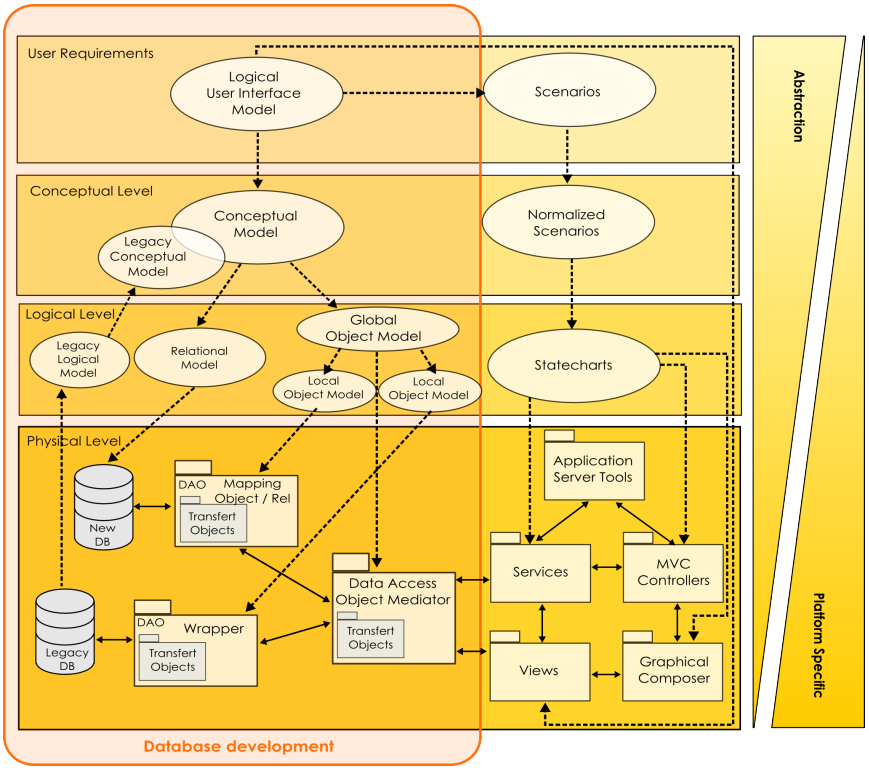


Figure 4.2: Overview of the ReQuest approach.

4.4 Overview of the approach

In the alternative *RAINBOW* approach, we want to keep the same overall philosophy while focusing on the specification of static data requirements as part of a greater Requirements Engineering process. The specificities of this approach lead us to specialise the techniques presented in Part I and integrate

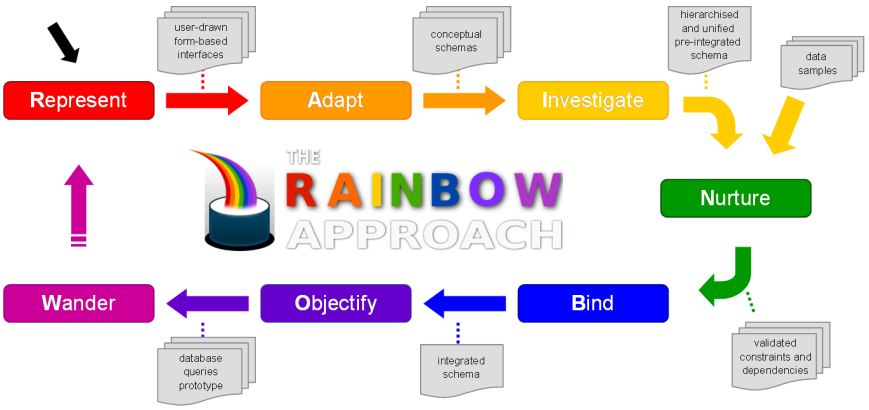


Figure 4.3: Overview of the RAINBOW approach.

them into a semi-automatic seven-step process (see Fig. 4.3) that does not aim to provide a ready-to-use application, but a set of specification documents and tools, in order to support the development of future applications and overcome the limitations synthesized in Part I:

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 this context, the development of an appropriate tool support is therefore a crucial and integral part of the approach. Besides, as one may notice, these seven steps address the various challenges regarding Database forward engineering, Database reverse engineering and Prototyping, which were presented in Part I. Indeed:

- Chapter 5 addresses the expression of requirements through user-drawn form-based interfaces (*Represent*).
- Chapter 6 explains how to translate these interfaces into data models (*Adapt*).
- Chapter 7 deals with the analysis of these data models to clarify terminological and structural ambiguities before performing a pre-integration of the models (*Investigate*).
- Chapter 8 engages in the elicitation of additional constraints and dependencies within the pre-integrated model (*Nurture*).
- Chapter 9 addresses the final schema integration and refinement of the pre-integrated data model (*Bind*).
- Chapter 10 presents the generation, integration and testing of applicative components based on the integrated data models, for the purpose of ultimately validating the elicited requirements (*Objectify* and *Wander*).

Subsequently, Chapter 11 discusses the dedicated tool support. In this doctoral research, we mainly focus on the five first steps of the approach, since the generation of the components is relatively straightforward and the manipulation of a reactive prototype mainly adds another level of validation.

Chapter 5

REPRESENT

Expressing concepts through form-based interfaces

In this chapter, we address the expression of requirements through form-based interfaces by end-users. Firstly, we recall the concerns regarding the existing UIDLs and how they lead us to define our own simplified form model. We then detail this simplified form model and how to use it in order to specify a set of form-based interfaces.

5.1 Concerns

The main concern about existing UIDLs comes from the structure of the language itself. Indeed, since they must express complex interfaces, layouts and behaviours, their XML structure becomes complex and difficult to read, and furthermore, the end-users may be overwhelmed by this superabundance of available widgets and compositions.

In the RAINBOW approach, we really want to focus on simple interface widgets that can allow end-users to simply express concepts, while casting away the technical aspects of layout. By defining a simple XML language to express the structure of our interfaces, the latter can be rendered later on in a more stylish way using templates, style sheets, eXtensible Stylesheet Language Transformations (XSLT) transformers and so on.

This should therefore help the *analysts* to draw the attention of end-users to the semantics of the used vocabulary (ambiguous terms, synonyms, recurring structures, ...) in order to build a clean and clear set of interfaces.

5.2 RAINBOW's Simplified Form Model

For this purpose, we propose the RAINBOW's Simplified Form Model (RSFM), based on the most usual form widgets, which can intuitively be mapped to the GER model. This model is intended to be transparently used by the end-users to express concepts, and includes information for designers and CASE developers that would like to instantiate or extend it. The RSFM foremost lays the foundation for an *exploratory* type of prototyping, though its definition also provides possible *evolutionary* perspectives.

To build the interface corresponding to each concept, we suggest to use a limited set of primitive widgets, which are simple but usual high-level form widgets through which any other widget can be expressed. They are classified as follows:

- containers: `forms`, `fieldsets` and `tables`;
- simple widgets: `inputs`, `selections` and `buttons`.

Consequently, the RFSM could be seen as a model halfway between UsiXML's *Abstract User Interface* (AUI) and *Concrete User Interface* (CUI) models [UsiXML, 2007]. Indeed, the RFSM actually defines a set of abstract containers and individual components having direct concrete widgets counterparts, which is usually not the case for UIDLs.

All these widgets may have specific properties, but they share the following properties:

- a unique and mandatory *identifier*;
- a mandatory *label*, which is the *visible* name of the widget. It may only be composed of letters, white spaces, dashes ('-') and numbers;
- an optional *term*, which is the *semantic* name of the widget, i.e. the non plural concept conveyed by the widget. If null, the term and the label are considered to be equivalent, but when analysing and unifying the terminology of the labels, it may be useful to differentiate them (e.g. the label of a `table` may be "Products", but its term would be "Product"). The term may only be composed of letters, white spaces, dashes ('-') and numbers;
- a *qualifier*, which is an optional additional piece of labelling used to prevent two widgets to have the exact same name when they have the same parent (e.g. in a `fieldset`, we may have two inputs labelled "Address", but respectively qualified as "Primary" and "Secondary"). The qualifier may only be composed of letters, white spaces, dashes ('-') and numbers. If it is not empty, the visible name of the widget will be "Label (qualifier)";

- an optional *description*, which can hold any relevant description, explanation or additional information regarding this element, its behaviour and/or the concept it conveys.

Besides, their layout is constrained, as will be exposed for each of them. In order to illustrate the elements and their properties, Fig. 5.1 provides an example of a simple form intended to gather information on a person.

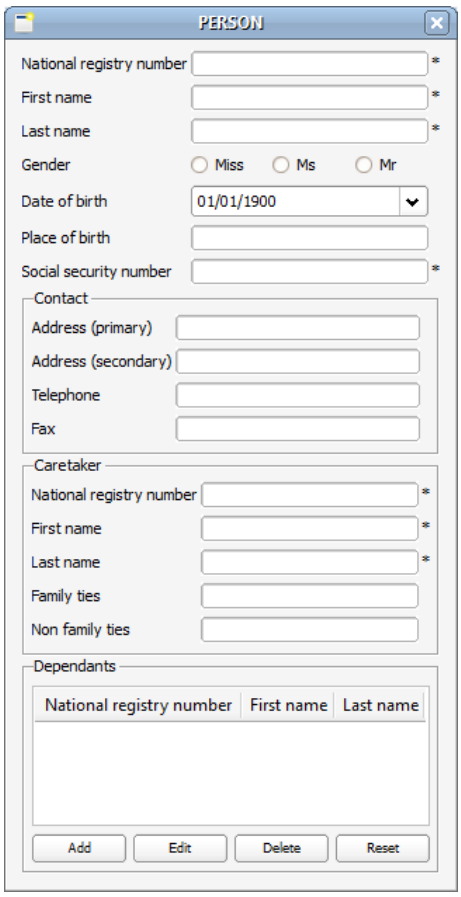


Figure 5.1: A simple form gathering information on a person.

For a given software engineering project using the RAINBOW approach, a set of forms containing other widgets will therefore be built according to the RSFM. A given form F will contain a set of widgets that can be filled with any content that does not contradict the various properties defined for the form and

its widgets. Each of these unique combinations of filling is called an *instance* of the form F.

Forms

A *form* is a top level container representing the concept of window (Fig. 5.2). It may contain any of the *other* elements and has the following properties:

- a mandatory *identifier*;
- a mandatory *label*;
- an optional *term*;
- an optional *qualifier*;
- an optional *description*.

Figure 5.2: A form widget.

Besides, additional information may be provided regarding the content of the elements the form contains when it is accepted:

- the *unique constraints* specify the different (sets of) mandatory widgets whose content form an identifier for the all the possible instances of the given form. Each identifier may be:
  - *primary*, which means that this set of widgets will be used as the main and preferred identifier for the concept conveyed by the form. For instance, the input labelled `National Registry Number` could be a primary identifier for the form `Person`;
  - *secondary*, which means that this set of widgets should not be regarded as the main or preferred way to identify the instances of the given form. For instance, the input labelled `Social Security Number` could be a secondary identifier for the same `Person` form, but would be less interesting than the `National Registry Number`;

- the *existence constraints* specify which restrictions should be imposed to the instances of (sets of) optional widgets of the form. These restrictions can be:
  - *coexistence*, which implies that all the widgets must be either empty or filled together. For instance, the inputs labelled `Date of birth` and `Place of birth` could be specified as coexistent;
  - *at-most-one*, which implies that there can be, at most, only one of the widgets that is filled. For instance, one could require a `Person` to fill exclusively the fieldset `Caretaker` or the table `Dependants` or none of them.;
  - *at-least-one*, which implies that at least one of the widgets should not be empty. For instance, in the fieldset `Contact`, one could require to fill at least one widget among "Address (primary)", "Address (secondary)", `Telephone` and `Fax`;
  - *exactly-one*, which implies that there must be one and only one of the widgets that is not be empty. For instance, in the fieldset `Caretaker`, one could require to specify either the `Family ties` or the `Non family ties`.

Fieldsets

A *fieldset* is a container used to group any other elements except forms (Fig. 5.3), like `Contact` and `Caretaker` in Fig. 5.1.

Figure 5.3: A fieldset widget.

A fieldset has the following properties:

- a mandatory *identifier*;
- a mandatory *label*;
- an optional *term*;
- an optional *qualifier*;
- an optional *description*;
- a mandatory *cardinality*, which defines if the widget if *optional* or *mandatory*. A mandatory fieldset requires to have at least one of its children

widgets that is not empty. For instance, if the fieldset `Contact` was mandatory, it would require at least one of the widgets among `Address (primary)`, `Address (secondary)`, `Telephone` and `Fax` to be filled;

- a mandatory *distinctiveness*, which defines if a given combination of children values must correspond to one and only one instance of the parent widget (*true*) or not (*false*). For instance, the fieldset `Contact` is not distinctive, because a given combination of `Address (primary)`, `Address (secondary)`, `Telephone` and `Fax` may correspond to several different `Persons`.
- optional *unique constraints*, as defined for the form widget;
- optional *existence constraints*, as defined for the form widget;
- an optional *prerequisite constraint*, which may specify, if the fieldset is optional, the identifiers of other widgets owned by the parent form. The specified widgets must be filled before the given widget can also be filled.

Tables

A *table* is a container used to structure elements sharing the same characteristics (Fig. 5.4), like `Dependants` in Fig. 5.1. It has the following properties:

- a mandatory *identifier*;
- a mandatory *label*;
- an optional *term*;
- an optional *qualifier*;
- an optional *description*;
- a mandatory *cardinality*, as defined for the fieldset widget;
- a mandatory *distinctiveness*, which defines if a given combination of children values for a given row must correspond to one and only one instance of the parent widget (*true*) or not (*false*);
- optional *unique constraints*, which specifies the different (sets of) mandatory widgets whose content form an identifier for the all the possible instances of rows for the given table, as defined for the form widget;
- optional *existence constraints*, which specifies which restrictions should be imposed to the instances of (sets of) optional widgets of rows for the given table, as defined for the form widget;
- an optional *prerequisite constraint*, as defined for the fieldset widget.

Among the widgets, the table is clearly the most complex one. Its columns can only host simple widgets, i.e. inputs, selections and buttons. In addition,

it is bundled with four buttons to add, edit, delete and reset the entries of the table. When adding or editing an entry, a form is automatically generated using the same graphical chart as the rest of widgets (Fig. 5.5). The motivation is to keep the table as a way to handle multiple occurrence of simple data structures, or as a "view" on more complex data structures that would require drawing additional forms to specify all their details.

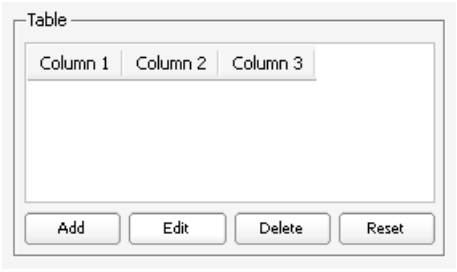


Figure 5.4: A table widget.

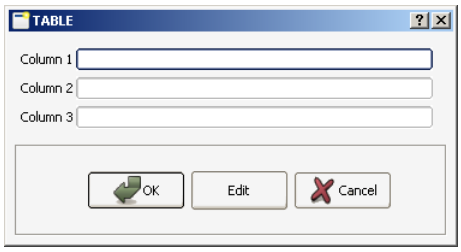


Figure 5.5: A simple form gathering information on a person.

Inputs

An *input* is a widget designed to receive simple textual input (Fig. 5.6).

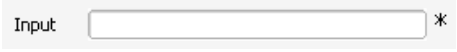


Figure 5.6: A mandatory input widget.

An input has the following properties:

- a mandatory *identifier*;
- a mandatory *label*;
- an optional *term*;

- an optional *qualifier*;
- an optional *description*;
- a mandatory *cardinality*, as defined for the `fieldset` widget;
- a mandatory *value type*, which indicates the expected type of the field. The value types are `text`, `integer`, `real`, `boolean` and `date`. For instance, the `First Name` would be a text, the `Social Security Number` would be an integer and the `Date of birth` would be a date. Setting the value type will restrict accordingly the characters that can be typed into the field;
- an optional *value size*, which indicates the expected size of the field, if relevant;
- an optional *formula*, which explains how the content of this widget should automatically be computed according to other variables. For instance, an `input` labelled `Age` could be automatically computed from another an `input` labelled `Date of birth`;
- an optional *prerequisite constraint*.

Selections

A *selection* is a widget designed to let the user choose zero, one or several values among a non empty set of predefined values (Fig. 5.7). This field can correspond to various combinations of widgets in GUIs, such as a group of radio buttons or checkboxes, a list of a combobox. If needed, users should be able to provide additional values to the predefined ones. It has the following properties:

- a mandatory *identifier*;
- a mandatory *label*;
- an optional *term*;
- an optional *qualifier*;
- an optional *description*;
- a mandatory *cardinality*, which defines if the minimum and maximum of items that may be selected. We restrict the possible cardinalities to the most common one, i.e. *at most one*, *exactly one*, *at least one*, *zero to many*. This cardinality must of course be consistent with the available number of selectable values.
- a mandatory *editability*, that specifies if the users can provide additional values to the predefined ones (*true*) or not (*false*);

- a mandatory *value type*, which indicates the expected type of the field when the selection is editable. The value types are `text`, `integer`, `real`, `boolean` and `date`;
- an optional *value size*, which indicates the expected size of the field, if the selection is editable;
- an optional *formula*;
- an optional *prerequisite constraint*.

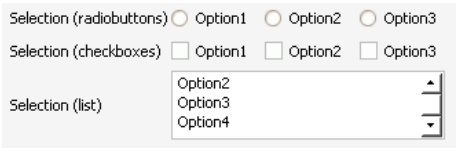


Figure 5.7: Different representations of the selection widget.

The selectable values are known as *options*, and have the following properties:

- a mandatory *identifier*;
- a mandatory *label*;
- an optional *term*;
- an optional *qualifier*;
- an optional *description*;

To simplify the use of the widget, we chose to make it automatically adapt its rendering according to the number of available options and the minimal and maximal cardinalities:

- a set of radio buttons, for 3 options or less, and a cardinality of at most one or exactly one;
- a set of checkboxes, for 3 options or less, and a cardinality of at least one or zero to many;
- a list for all other cases.

Buttons

A *button* is a widget that allows to specify a set of *actions* that must be triggered when the widget is pressed (Fig. 5.8).

A button has the following properties:

- a mandatory *identifier*;
- a mandatory *label*;

Figure 5.8: An button widget.

- an optional *term*;
- an optional *qualifier*;
- an optional *description*;
- an optional *prerequisite constraint*.

Each *action* attached to a given button must explain through a *description* what happens when the button is pressed.

Getting and setting the properties of the widgets

For further processing by eventual analysts and developers, all the properties expressed for the different types of widgets must be accessible in read and write mode. For a given widget w, one can get and set a property using the following functions;

- `get<PropertyName>`(w)
- `set<PropertyName>`$(w, value)$

For instance, we can get the label of a table t using `getLabel`(t), and set it to "Dependants" using `setLabel`$(t$,"Contact").

We call $\mathcal{F}_\mathcal{E}$ the set of all the possible functions available for the different types of widgets. A function $f \in \mathcal{F}_\mathcal{E}$ should return $f(w)$ =`null` if it is not defined for the type of widget w.

Tree structure

Similarly, analysts and developers can take advantage of the inherent tree-like structure of the RSFM, which enables us to the direct parent and children of each widget, as well as their ancestors and descendants. For a given widget w, they are accessible using:

- `getDirectParent`(w), which returns `null` if w is a form and the parent widget otherwise;
- `getDirectChildren`(w), which returns $\emptyset$ if w is a simple widget (i.e. an `input`, a `selection` or a `button`) or the set of direct children if w is a container (i.e. an `form`, a `fieldset` or a `table`).

Fig 5.9 illustrates the tree-like structure of the `Person` form shown in Fig. 5.1.

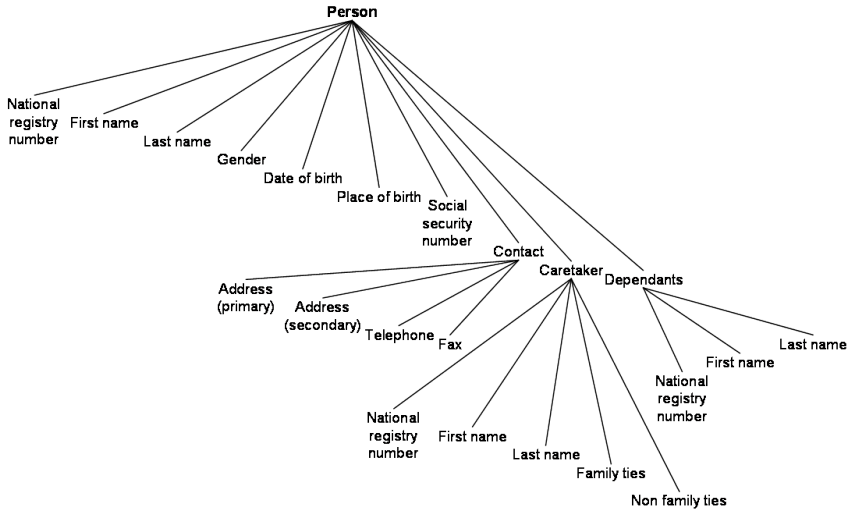


Figure 5.9: The tree-like structure of the `Person` form shown in Fig. 5.1.

5.3 Managing the process

During the *Represent* step, the end-users are invited to draw and specify a set of form-based interfaces to perform usual tasks of their application domain. In this section, we present a series of recommendations to efficiently prepare and execute this process.

5.3.1 Preparation guidelines for the analyst

The first step in the process is twofold. On the one hand, the participants must get together and discuss the objectives and organisation of their project. On the other hand, the analysts need to understand the application domain, and the subject of the software engineering for which the RAINBOW approach will be used. Since the approach is not intended to replace any existing approach, but rather complement them, traditional elicitation techniques can be used for this purpose.

Meeting and choosing the participants

From the start, the analysts therefore need to meet with as many stakeholders as possible, and carefully negotiate the choice of user participants (as explained in Chapter 3.7), since it is not systematically possible for all those affected by

the design effort to fully participate in the process. Preferably, one will choose among experienced volunteer users.

Planning the project

Once the participant end-users are selected, the planning for the execution of the RAINBOW approach should be defined. This planning may evidently evolve later on, but an overall estimation of the duration of the tasks should be done, and the appointments should be set for the training sessions.

Training the end-users

The last step before starting the drawing is to train the participant end-users. This involves explaining the overall philosophy of the approach and familiarising them with the available widgets, their specification and the tool-support that they will use to draw relevant form-based interfaces.

Getting a preliminary insight of the application domain

In parallel, using *interviews*, the analysts should be able to draw a first outline of the project and have a basic understanding of the subject, its main objectives and possible challenges and conflicts. Any existing *documentation* on the application domain and its process, as well as possible paper forms and existing applications should be studied and made available for further reference. Similarly, listing the expected tasks the future system should support, and defining their associated *use cases* [OMG, 2007], is helpful to direct the end-users during their drawing.

5.3.2 Execution guidelines and recommendations

Once the preparation step is over, the drawing step may begin. The *end-users* are invited to *draw form-based interfaces* to describe the key concepts of their application domain and enabling them too perform simple and usual tasks, such as a window to introduce a new registered customer into an hypothetical system. They must provide details on the concepts through the previously defined properties of the widgets. Based on the preliminary analysis led during the preparation step, the *analysts* can advantageously guide the end-users by precisely suggesting them key concepts and tasks, and assist them to use the RFSM in their drawing work.

Let us keep in mind that the objective here is *not* to lead the end-users to draw the interfaces of a future application, but to express requirements through

a medium that is familiar to them. To produce a set of form-based interfaces $\mathcal{I} = \{interface_1, ..., interface_n\}$ agreeing with the RAINBOW approach, we recommend to respect the following recommendations:

- Respect the structure of the RSFM;
- Whenever possible, provide the maximum information for each interface element, even if the latter is optional;
- Regarding the labelling the interface elements, in order to ensure expressiveness (and to ease the *Investigation* phase):
  - one should use a structured concatenation of words, numbers and separators (such as white spaces, commas, ...) rather than just any series of characters;
  - abbreviations and acronyms should be avoided;
  - labels should be wisely and consistently chosen, typically to limit the risks of synonymy and polysemy;
  - two elements at the root of the same container (`form`, `fieldset` or `table`) should not have the exactly same label: if necessary, use a qualifier to differentiate the elements.

In Chapter 7, we will introduce mechanisms to particularly help unifying the terminology of the widgets from the start, during this drawing phase.

5.3.3 Assisting the end-users through the tool support

The drawing phase should be performed using a dedicated drawing tool. For this purpose, a proof-of-concept tool-support will be presented in Chapter 11. Besides, we already mentioned that the analyst should assist the end-users throughout this representation phase, in order to limit the risk of ambiguities and inconsistencies.

5.4 Output

The output of the Represent phase is therefore a set of form-based interfaces $\mathcal{I} = \{interface_1, ..., interface_n\}$ using elements of the RSFM, and representing various concepts that need to be analysed. We call $\mathcal{E}_\mathcal{I}$ the set of all the widgets used in $\mathcal{I}$.

5.5 A running example

Let us consider a simple running example to illustrate the process: the context is the development of a tailored IT solution to manage a small company that offers `Services` and sales `Products`, including `Special goods`, through different `Shops`. They wish to store information on their `Providers` and `Customers`, including the `Orders` that they submitted. Fig. 5.10 illustrates forms that the end-users might draw for this purpose. For instance, for each customer, personal information including his main and alternative addresses are stored, as well as the list of orders that he issued. Each of these orders mention information on the context of its creation, and list the associated list of products, and so on. For the sake of further discussion, we consider that the forms do not mention any unique, existence or prerequisite constraint.

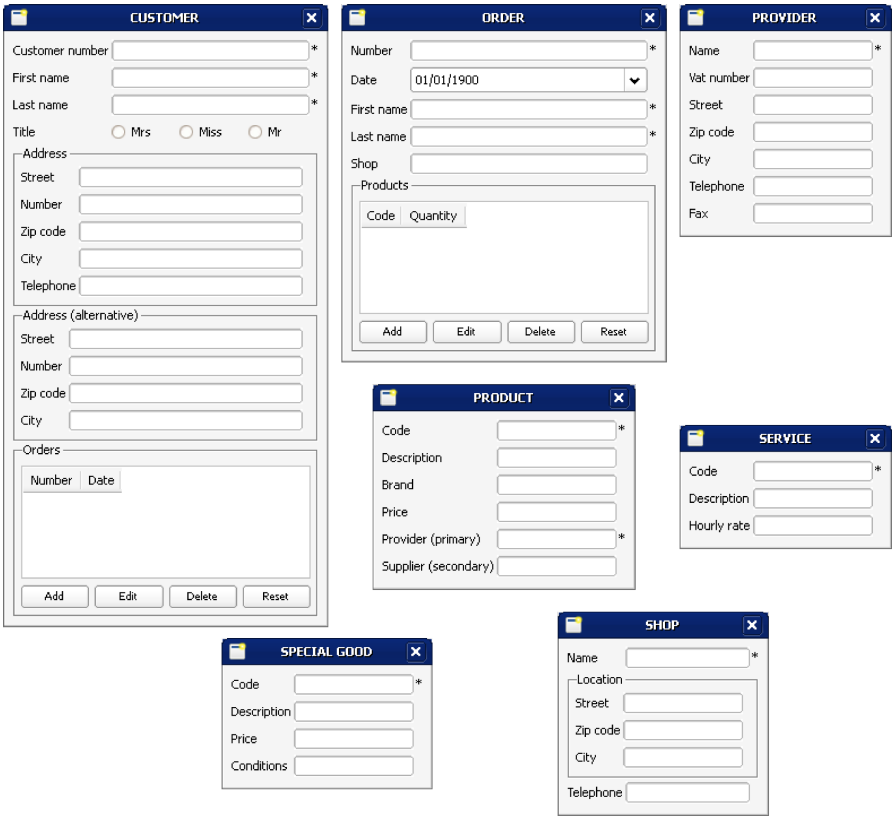


Figure 5.10: Possible user-drawn form-based interfaces for the management of a small company that offers services and sales products.

Chapter 6

ADAPT

Extracting data models from form-based interfaces

In this chapter, we address the automatic twofold translation of the previously drawn form-based interfaces into their corresponding logical data model that expresses its underlying data structure using the GER model, which paves the way to using the transformational power of conceptual modelling later on. For this purpose, we introduce the intuitive mapping rules that we use, then formalise them into two consecutive algorithms to first perform a *raw* transformation, then a *refined* transformation.

6.1 Intuitive mapping between the RSFM and the GER model

As explored in [Choobineh et al., 1992] or [Rollinson and Roberts, 1998], it is possible to translate a form based-interface into a logical user interface model that expresses its underlying data structure in an abstract way. We choose to use the GER model, which was presented in Section 2.2.3 and is a wide-spectrum variant of the popular Entity-relationship model that encompasses logical and conceptual structures [Hainaut, 2005]. We can intuitively present the mapping rules that we want to define as follows, and illustrate them for a simple form in Fig. 6.1:

- we create one *schema* per *form*, then, within this schema:
- each *form* is mapped to an *entity type*;
- each *fieldset* is mapped to a single-valued *compound attribute*;
- each *table* is mapped to a multi-valued *compound attribute*;

- each *input* is mapped to a single-valued *simple attribute*;
- each *selection* is mapped to a *simple attribute* with a predefined domain of values;
- each *button* is mapped to an *procedural unit*.

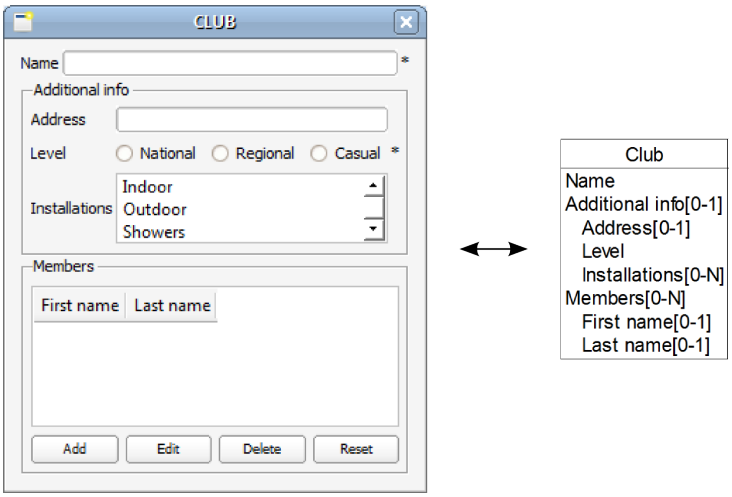


Figure 6.1: Illustration of the intuitive mapping rules for a simple form.

The resulting *raw* schemas contains only monolithic entity types, with possibly several levels of compound attributes. However, we would like to work with *refined* schemas containing "flat" entity types (i.e. entity types having only simple attributes), in order to ease the handling of the next steps of the approach. To obtain these simple "refined" entity types, the idea is to recursively transform the compound attributes into entity types. Let us now formalise both the raw and refined transformation process.

6.2 Raw transformation

The intuitive raw mapping rules of the previous section can be formalised as follows. Given a set of form-based interfaces $\mathcal{I} = \{interface_1, ..., interface_n\}$, using a set $\mathcal{E}_\mathcal{I}$ of interface elements, we create a corresponding set of schemas $\mathcal{S} = \{schema_1, ..., schema_n\}$ containing the data objects $\mathcal{E}_\mathcal{S}$.

To achieve this, Algorithm 6.1 creates a $schema_i$ for each $interface_i$, then creates an $entityType_i$ within this schema. Then, it uses Algorithm 6.2 to create its structure by creating the logical counterpart of each interface $element \in \mathcal{E}_\mathcal{I}$ into its logical $parentdataobject$.

Algorithm 6.1 Adapt : extract the data model of each interface contained in the set $\mathcal{I}$

Require: $\mathcal{I} = \{interface_1, ..., interface_n\} \wedge n > 0$
Ensure: $\mathcal{S} = \{schema_1, ..., schema_n\} \wedge \ \forall i \in [1, n] : \exists entityType_i \in schema_i$

1: **procedure** REPRESENT($\mathcal{I}$)
2: $\mathcal{S} \leftarrow \emptyset$
3: **for all** $interface_i \in I$ **do**
4: $schema_i \leftarrow$ createSchema($\mathcal{S}$)
5: $id_i \leftarrow$ getId($interface_i$)
6: $label_i \leftarrow$ getLabel($interface_i$)
7: $term_i \leftarrow$ getTerm($interface_i$)
8: **if** $term \neq$ `null` **then**
9: $term \leftarrow label$
10: **end if**
11: $qualifier_i \leftarrow$ getQualifier($interface_i$)
12: $description_i \leftarrow$ getDescription($interface_i$)
13: $name_i \leftarrow label_i$
14: **if** $qualifier_i \neq$ `null` **then**
15: $name_i \leftarrow name_i +' ('+qualifier_i+')'$
16: **end if**
17: $entityType_i \leftarrow$ createEntityType($schema_i, name_i$)
18: $directChildren_i \leftarrow getDirectChildren(interface_i)$
19: **for all** $widget_j \in directChildren_i$ **do**
20: representChild($widget_j, entityType_i$) ▷ See Algorithm 6.2 on page 84
21: **end for**

22: $uniqueconstraints_i \leftarrow$ getUniqueConstraints($interface_i$)
23: **for all** $uniqueconstraint_{i_j} \in uniqueconstraints_i$ **do**
24: $uniquetype_{i_j} \leftarrow getUniqueType(uniqueconstraint_{i_j})$
25: $uniqueids_{i_j} \leftarrow getUniqueIds(uniqueconstraint_{i_j})$
26: addUniqueConstraint($entityType_i, uniquetype_{i_j}, uniqueids_{i_j}$)
27: **end for**

28: $existenceconstraints_i \leftarrow$ getExistenceConstraints($interface_i$)
29: **for all** $existenceconstraint_{i_j} \in I$ **do**
30: $existencetype_{i_j} \leftarrow getExistenceType(existenceconstraint_{i_j})$
31: $existenceids_{i_j} \leftarrow getExistenceIds(existenceconstraint_{i_j})$
32: addExistenceConstraint($entityType_i, existencetype_{i_j}, existenceids_{i_j}$)
33: **end for**
34: storeMetaProperties($entityType_i$, id_i, $label_i$, $term_i$, $qualifier_i$, $description_i$)
35: **end for**
36: **end procedure**

Algorithm 6.2 AdaptChildInto : extract the data object corresponding to a given widget into the given parent data object (1/3)

Require: $widget \neq \emptyset \wedge widget \in \mathcal{E}_{\mathcal{I}} \wedge parentdataobject \neq \emptyset \wedge parentdataobject \in \mathcal{E}_{\mathcal{S}}$
$\wedge$ getType($parentdataobject$) $\in \{\texttt{ENTITYTYPE}, \texttt{COMPOUNDATTRIBUTE}\}$
Ensure: $dataobject \neq \emptyset$
$\wedge$ getParent($dataobject$) = $parentdataobject$

```
 1: procedure REPRESENTCHILD(widget, parentdataobject)
 2:     id ← getId(widget)
 3:     label ← getLabel(widget)
 4:     term ← getTerm(widget)
 5:     if term ≠ null then
 6:         term ← label
 7:     end if
 8:     qualifier ← getQualifier(widget)
 9:     description ← getDescription(widget)
10:     cardinality ← getCardinality(widget)
11:     name ← label
12:     if qualifier ≠ null then
13:         name ← name +' ('+qualifier+')'
14:     end if
15:     if widgetType = FIELDSET ∨ widgetType = TABLE then
16:         if cardinality = optional then
17:             minCard ← 0
18:         else
19:             minCard ← 1
20:         end if
21:         if widgetType = FIELDSET then
22:             maxCard ← 1
23:         else ▷ widgetType = TABLE
24:             maxCard ← N
25:         end if
26:         dataobject ← createCompoundAttribute(parentdataobject, name,
    minCard, maxCard)
27:         directChildren ← getDirectChildren(widget)
28:         for all widget_j ∈ directChildren do
29:             representChild(widget_j, dataobject)
30:         end for
31:         distinctiveness ← getDistinctiveness(widget)
32:         uniqueconstraints ← getUniqueConstraints(widget)
33:         for all uniqueconstraint_i ∈ uniqueconstraints do
34:             uniquetype_i ← getUniqueType(uniqueconstraint_i)
35:             uniqueids_i ← getUniqueIds(uniqueconstraint_i)
36:             addUniqueConstraint(dataobject, uniquetype_i, uniqueids_i)
37:         end for
```

▷ [continued on page 85]

Algorithm 6.3 AdaptChildInto (2/3)

```
38:         existenceconstraints ← getExistenceConstraints(widget)
39:         for all existenceconstraint_i ∈ existenceconstraints do
40:             existencetype_i ← getExistenceType(existenceconstraint_i)
41:             existenceids_i ← getExistenceIds(existenceconstraint_i)
42:             addExistenceConstraint(dataobject, existencetype_i, existenceids_i)
43:         end for
44:         prerequisiteids ← getPrerequisiteConstraint(widget)
45:         storeMetaProperties(dataobject, id, label, term, qualifier, description,
    distinctiveness, prerequisiteids)
46:     else if widgetType = INPUT ∨ widgetType = SELECT then
47:         if widgetType = INPUT then
48:             if cardinality = optional then
49:                 minCard ← 0
50:             else
51:                 minCard ← 1
52:             end if
53:             maxCard ← 1
54:         else ▷ widgetType = SELECT
55:             if cardinality = at most one then
56:                 minCard ← 0
57:                 maxCard ← 1
58:             else if cardinality = exactly one, then
59:                 minCard ← 1
60:                 maxCard ← 1
61:             else if cardinality = at least one then
62:                 minCard ← 0
63:                 maxCard ← N
64:             else
65:                 minCard ← 1
66:                 maxCard ← N
67:             end if
68:         end if
69:         valueType ← getValueType(widget)
70:         valueSize ← getValueSize(widget)
71:         dataobject ←createSimpleAttribute(parentdataobject, name, minCard,
    maxCard, valueType, valueSize)
72:         formula ←getFormula(widget)
73:         prerequisiteids ←getPrerequisiteConstraint(widget)
74:         if widgetType = INPUT then
75:             storeMetaProperties(dataobject, id, label, term, qualifier,
    description, formula, prerequisiteids)
```

▷ [continued on page 86]

Algorithm 6.4 AdaptChildInto (3/3)

```
76:         else ▷ widgetType = SELECT
77:             editability ←getEditability(widget)
78:             options ←getOptions(widget)
79:             for all option ∈ options do
80:                 value ←getValue(option)
81:                 addValueConstraint(dataobject, value)
82:             end for
83:             storeMetaProperties(dataobject, id, label, term, qualifier,
    description, editability, formula, prerequisiteids)
84:         end if
85:     else ▷ widgetType = BUTTON
86:         proceduralunit ←createProceduralUnit(parentdataobject, name)
87:         actionDescriptions ← ∅
88:         actions ←getActions(widget)
89:         for all action ∈ actions do
90:             actionDescription ←getDescription(action)
91:             actionDescriptions ← actionDescriptions ∪ actionDescription
92:         end for
93:         storeMetaProperties(dataobject, id, label, term, qualifier, description,
    actionDescriptions)
94:     end if
95: end procedure
```

For this purpose, it uses the functions of $\mathcal{F}_{\mathcal{E}}$, as defined in Section 5.2. It also stores several meta properties into the logical elements, in order not to lose any semantic information and to ensure traceability.

When applied to the form-based interfaces of Fig. 5.10, we obtain the schemas and entity types depicted in Fig. 6.2. As we can for instance see, the form `Customer` is mapped to an entity with the same name, while the input `First name` is mapped to an atomic attribute `First name`, the fieldset `Address` is mapped to a compound attribute `Address`, and so on.

6.3 Refined transformation

At this point, each interface of $\mathcal{I}$ is mapped to an entity type of one of the raw schemas of $\mathcal{S}$. Given the tree-like structure of the interfaces, the corresponding entity types may contain compound attributes. However, in this doctoral research, we want to work with a sub-model of the GER restricted to "flat" entity types (i.e. entity types having only atomic attributes), binary relationship types (i.e. relationship types having exactly two roles) and IS-A hierarchies. This restriction does not reduce the impact of the approach, since more

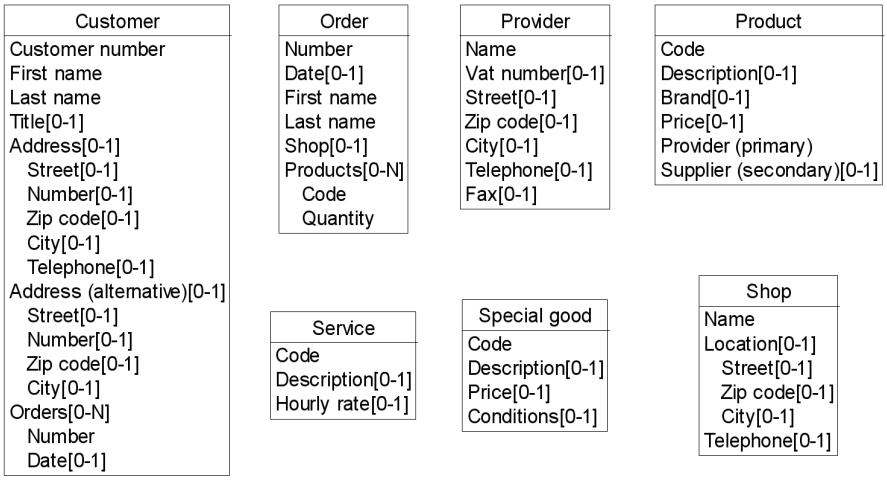


Figure 6.2: Translation of the interfaces into raw entity types.

complex structures can be converted into the latter using semantic-preserving transformations.

In order to "flatten" the entity types, we use Algorithm 6.5 to recursively extract each compound attribute from each individual entity type into another entity type, which is an equivalence preserving transformation. The original compound attributes are therefore "replaced" by a role referencing the newly created entity type using the metaproperty `targetEntityType`.

We call $\mathfrak{E}_F$ the set of entity types corresponding to a given form F. The complete data structure extraction of Fig. 5.10 therefore leads to the schemas and entity types illustrated in Fig. 6.3.

6.4 Managing the process and output

This automatic process requires no input from the end-users, and produces a set of schemas $\mathcal{S} = \{schema_1, ..., schema_n\}$ based on $\mathcal{I} = \{interface_1, ..., interface_n\}$. Given the nature of the process, there is a *injection* between $\mathcal{E}_\mathcal{I}$ and $\mathcal{E}_\mathcal{S}$, so that any interface widget can be mapped to a unique logical counterpart that is an entity type or a simple attribute, and conversely, any entity type or simple attribute of the schemas can be mapped to a unique widget.

Algorithm 6.5 Unfold : Adapt the schema by recursively transforming each compound attributes

Require: $entityType \neq \emptyset$
$\wedge\ entityType \in schema$
Ensure: $\nexists\ e \in schema$: getType(e) = `COMPOUNDATTRIBUTE`

```
1: procedure UNFOLD(entityType)
2:     for all element e_i ∈ entityType do
3:         if getType(e_i) = COMPOUNDATTRIBUTE then
4:             minCard1 ← getMinimumCardinality(e_i)
5:             maxCard1 ← getMaximumCardinality(e_i)
6:             isDistinctive ← getMetaProperty(e_i, distinctiveness)
7:             if thenisDistinctive = true
8:                 minCard2 ← 1
9:                 maxCard2 ← 1
10:            else
11:                minCard2 ← 0
12:                maxCard2 ← N
13:            end if
14:            id_i ← getMetaProperty(e_i, id)
15:            label_i ← getMetaProperty(e_i, label)
16:            term_i ← getMetaProperty(e_i, term)
17:            qualifier_i ← getMetaProperty(e_i, qualifier)
18:            description_i ← getMetaProperty(e_i, description)
19:            name_i ← getName(e_i)
20:            entityType_i ← createEntityType(schema, name_i)
21:            storeMetaProperties(entityType_i, id_i, label_i, term_i, qualifier_i,
    description_i)
22:            removeAttribute(entityType, e_i)
23:            relType_i ← createRelationshipType(entityType, entityType_i)
24:            setCardinalities(relType_i, minCard1, maxCard1, minCard2,
    maxCard2)
25:            role_i ← getRole(entityType, relType_i)
26:            setMetaProperty(role_i, entityType_i, targetEntityType)
27:            unfold(entityType_i)
28:        end if
29:    end for
30: end procedure

31: procedure ADAPTSCHEMA(schema)
32:    for all entityType ∈ schema do
33:        unfold(entityType)
34:        markAsRoot(entityType)
35:    end for
36: end procedure
```

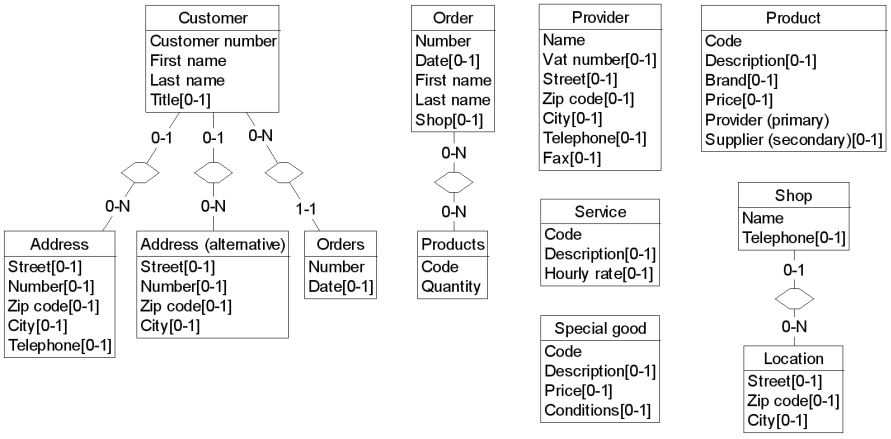


Figure 6.3: Translation of the raw entity types into independent schemas.

Chapter 7

INVESTIGATE

Analysing semantic and structural redundancies to manage commonality

Human-computer interfaces offer several levels of communication. For instance, Nielsen identifies seven layers, among which analysing the static appearance of a set of interfaces notably involves the *semantics* (the meaning of the expected interactions), *syntax* (the anticipated sequence of necessary actions to perform a task), *lexical* (the interaction tokens, such as labels and icons) and *alphabetic* (the primitive information units, such as letters, digits and colours) layers [Nielsen, 1986]. So many levels may obviously induce equivocal interfaces open to interpretation.

In particular, cross-analysing the individual schemas obtained by adapting the user-drawn forms usually brings to light possible ambiguities as well as redundant information contained in each interface. If these redundancies can be automatically identified and manually validated, they can be resolved afterwards in order to synthesize and refine an integrated conceptual schema. Typically, whereas we can observe that the constructs used by a single user are relatively (though not necessarily) consistent among the interfaces he draws, when considering multiple users, we notice that typical variabilities, ambiguities and redundancies may occur. These phenomena may concern the various properties of the widgets, such as the labels, qualifiers, descriptions, minimal cardinality, maximal cardinality, value type, value size, or domain of values.

In the scope of this doctoral research, we focus on the widgets labels to track down semantic and structural ambiguities, but the definitions and strategies that we propose could intuitively be extended to take in account other

properties. In this chapter, we therefore formalise the definition of semantic and structural similarity based on the labels of the widgets, then explain how to discover and present them for end-user arbitration and further processing.

7.1 Terminological ambiguities

As explained in Section 3.4.1, terminological ambiguities occur when elements of the schema are *semantically similar*, i.e. their name appear to be orthographically and/or ontologically similar. In our running example, the labels "Orders" and "Order" are orthographically similar, while the labels "Provider" and "Supplier" are ontologically similar, as illustrated in Fig. 7.1.

In this section, we therefore formalise these notions of semantic, orthographic and ontological similarities, then provide a strategy to discover such similarities among the elements of the underlying schema of the forms before presenting them to the end-users for validation.

7.1.1 Formalising the notions of similaritiy

Orthographic and ontological similarity for two strings

As explained in Section 3.4.1, String Metrics can be used to compare strings according to their spelling. We can therefore define the *orthographic similarity of two strings* as follows:

Definition 7.1. Given a String distance metric sdm and threshold t, we define two strings l_1 and l_2 as *orthographically similar* with respect to sdm and t, iff $sdm(l_1, l_2) \leq t$. ⌟

Besides, we also explained that Ontologies can be useful to track down similarities of meaning among a set of words. They offer many information on the concepts and their relations within a given domain, but in our research we focus mainly on the proximity of meaning. In particular, for a given ontology $\mathcal{O}$, we consider the synonyms function $s_{\mathcal{O}}$ that associates a given string l to its set of synonyms in $\mathcal{O}$, if any. For any $l' \in s_{\mathcal{O}}(l)$, we naturally have $l \in s_{\mathcal{O}}(l')$.

Definition 7.2. Given an ontology $\mathcal{O}$ and a synonyms function $s_{\mathcal{O}}$, we define two strings l_1 and l_2 as *ontologically similar* with respect to $\mathcal{O}$ and $s_{\mathcal{O}}$, iff : $l_1 \in s_{\mathcal{O}}(l_2)$ ⌟

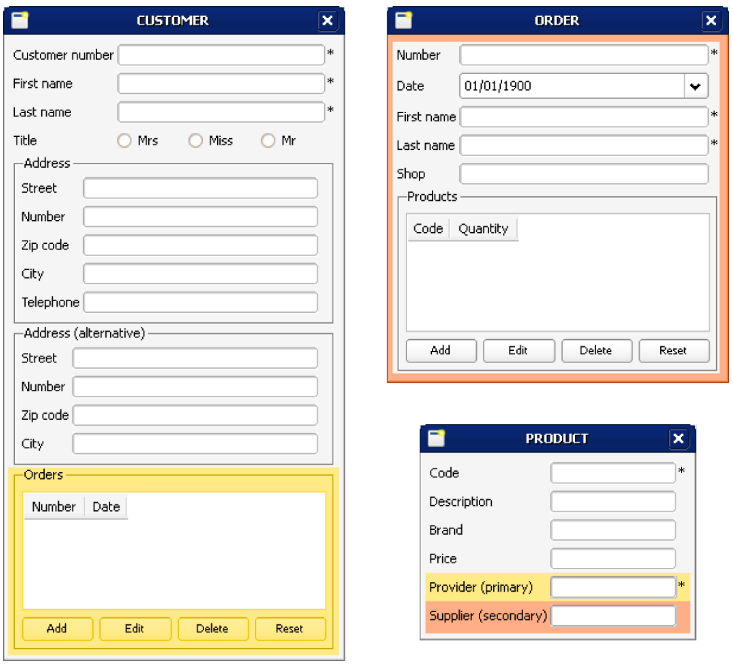


Figure 7.1: Highlighting of the terminological ambiguities for the labels "Orders" and "Order" (which are orthographically similar), and the labels "Provider" and "Supplier" (which are ontologically similar).

Semantic similarity for two word-based terms

Though interface widgets are associated with visible labels, we are truly interested in the *terms* hiding behind the labels, since they represent the semantic concepts conveyed by the widgets. Besides, if we recall the mapping rules of Chapter 6, we know that if no specific term has been used to describe a widget, the label is used as the default term. Let us therefore focus on the semantic similarity of the terms used in the set $\mathcal{I}$ of form-based interfaces, and by extension, their underlying set $\mathcal{S}$ of schemas.

Given the recommendations of the Represent phase (see Section 5.3.2), we can reasonably assume that the terms (and labels) used by end-users in $\mathcal{I}$ are not just arbitrary series of characters, but a systematic and structured concatenation of words, numbers and separators (such as white spaces, commas, ...). This implies that two terms may not be orthographically or ontologically similar *as such*, but that the words composing them could be.

Recall for instance the strings "Primary provider" and "Alternative supplier" (see Section 3.4.1), that could be used as realistic terms in form-based

interfaces. They are not orthographically similar as such, and since they each consist of the combination of an adjective and a noun, they can't be found as such in an ontology. This means that they won't produce any synonyms, and consequently, they can't be be ontologically similar either. However, if we consider the words composing them, we can find that "provider" and "supplier" are synonyms, which should lead us to conclude that "Primary provider" and "Alternative supplier" do seem semantically similar.

In order to improve the analysis of orthographic and ontological similarities among the terms of $\mathcal{I}$, we therefore need to break these terms into words in order to consider them too. For this purpose, we consider the function w which returns the set of words contained by a given string, typically by using the space character as a separator. For instance, we have $w(\text{“product”}) = \{\text{“product”}\}$ and $w(\text{“first name”}) = \{\text{“first”},\text{“name”}\}$.

Intuitively, we will therefore consider two terms t_1 and t_2 as semantically similar if:

- t_1 and t_2 are orthographically or ontologically similar as such, or
- t_1 is orthographically or ontologically similar to at least one of the words of t_2, or
- t_2 is orthographically or ontologically similar to at least one of the words of t_1
- at least one of the words of t_1 is orthographically or ontologically similar to at least one of the words of t_2.

In other words:

Definition 7.3. Given a string distance metric *sdm* and its associated threshold t_{sdm}, an ontology $\mathcal{O}$ and its synonyms function $s_{\mathcal{O}}$, we define two terms t_1 and t_2 as *semantically similar* with respect to *sdm*, t_{sdm}, $\mathcal{O}$ and $s_{\mathcal{O}}$ (noted $t_i \overset{sdm,t_{sdm},\mathcal{O},s_{\mathcal{O}}}{\Longleftrightarrow} t_j$), iff:

t_1 and t_2 are *orthographically similar* with respect to *sdm* and t_{sdm}

$\vee$ t_1 and t_2 are *ontologically similar* with respect to $\mathcal{O}$ and $s_{\mathcal{O}}$

$\vee$ $\exists w_{2_i} \in w(t_2)$: t_1 and w_{2_i} are *orthographically similar* with respect to *sdm* and t_{sdm}

$\vee$ $\exists w_{2_i} \in w(t_2)$: t_1 and w_{2_i} are *ontologically similar* with respect to $\mathcal{O}$ and $s_{\mathcal{O}}$

$\vee$ $\exists w_{1_i} \in w(t_1)$: t_2 and w_{1_i} are *orthographically similar* with respect to *sdm* and t_{sdm}

$\vee\ \exists w_{1_i} \in w(t_1)\ :\ t_2$ and w_{1_i} are *ontologically similar* with respect to $\mathcal{O}$ and $s_{\mathcal{O}}$

$\vee\ \exists w_{1_i} \in w(t_1), w_{2_i} \in w(t_2)\ :\ w_{1_i}$ and w_{2_i} are *orthographically similar* with respect to sdm and t_{sdm}

$\vee\ \exists w_{1_i} \in w(t_1), w_{2_i} \in w(t_2)\ :\ w_{1_i}$ and w_{2_i} are *ontologically similar* with respect to $\mathcal{O}$ and $s_{\mathcal{O}}$

⌟

Semantic similarity for a set of word-based terms

The definition of semantic similarity for two terms leads in turn to the definition of equivalence classes within a given set of terms $\mathscr{T}$, which we call*semantically similar subsets* of $\mathscr{T}$. In such a subset $\mathcal{T}_i$, each term is close enough from at least one of the other terms, while being far enough from any term of another subset $\mathcal{T}_j$. More formally:

Definition 7.4. Given a string distance metric sdm and its associated threshold t_{sdm}, an ontology $\mathcal{O}$ and its synonym function $s_{\mathcal{O}}$, the set $\mathcal{T}_i = \{t_{i_1}, t_{i_2}, ..., t_{i_m}\} \subseteq \mathscr{T} = \{t_1, t_2, ..., t_n\}$ is a *subset of semantically similar terms* of $\mathscr{T}$, with respect to sdm, t_{sdm}, $\mathcal{O}$ and $s_{\mathcal{O}}$, iff:

$$\begin{aligned} &(n = 1\ \vee\ \forall\ t_{i_j} \in \mathcal{T}_i, \exists t_{i_k} \neq t_{i_j} \in \mathcal{T}_i && : t_{i_j} \xLeftrightarrow{sdm, t_{sdm}, \mathcal{O}, s_{\mathcal{O}}} t_{i_k}) \\ &\wedge(\forall\ t_{i_j} \in \mathcal{T}_i, \nexists t_k \in \mathscr{T} \backslash \mathcal{T}_i && : t_{i_j} \xLeftrightarrow{sdm, t_{sdm}, \mathcal{O}, s_{\mathcal{O}}} t_k) \end{aligned}$$

⌟

The set $\mathcal{T}_{\mathscr{T}}$ that contains all the $\mathcal{T}_i$ forms a partition of $\mathscr{T}$, so that:

- $\forall i \neq j : \mathcal{T}_i \cap \mathcal{T}_j = \emptyset$
- $\bigcup_i \mathcal{T}_i = \mathscr{T}$

Consider for instance the set of terms $\mathscr{T}$ = {“First Name”, “Last Name”, “Primary provider”, “Alternative supplier”, “Address” }. The subsets $\mathcal{T}_1$ = {“First Name”, “Last Name” }, $\mathcal{T}_2$ = {“Primary provider”, “Alternative supplier”} and $\mathcal{T}_3$ = { “Address” } are *semantically similar subsets* of $\mathscr{T}$ and form a partition for it.

7.1.2 Discovering terminological ambiguities

Building a thesaurus

In order to identify the semantically similar subsets of all the terms used in the set of schemas $\mathcal{S}$ obtained through the Adapt phase, we start by building a *thesaurus*. A thesaurus holds the mappings between terms, labels, qualifiers and identifiers within a set of schemas. More formally:

Definition 7.5. For a given set of schemas $\mathcal{S} = \{schema_1, ..., schema_n\}$, a *thesaurus* $\tau_{\mathcal{S}}$ contains mappings $l_{\tau_{\mathcal{S}}}$ and $q_{\tau_{\mathcal{S}}}$ so that:

- $l_{\tau_{\mathcal{S}}}(term) = \{(label_i, \{id_{i_j}\}) \mid \forall\, i, j\ \exists\ element_{i,j,k} \in schema_k$:

$$\begin{aligned} getId(element_{i,j,k}) &= id_{i_j} \\ getTerm(element_{i,j,k}) &= term \\ getLabel(element_{i,j,k}) &= label_i\} \end{aligned}$$

- $l_{\tau_{\mathcal{S}}}(term, label) = \{id_i \mid \forall\ i\ \exists\ element_{i,j} \in schema_j$:

$$\begin{aligned} getId(element_{i,j}) &= id_i \\ getTerm(element_{i,j}) &= term \\ getLabel(element_{i,j}) &= label\} \end{aligned}$$

- $q_{\tau_{\mathcal{S}}}(term) = \{(qualifier_i, \{id_{i_j}\}) \mid \forall\, i, j\ \exists\ element_{i,j,k} \in schema_k$:

$$\begin{aligned} getId(element_{i,j,k}) &= id_{i_j} \\ getTerm(element_{i,j,k}) &= term \\ getQualifier(element_{i,j,k}) &= qualifier_i\} \end{aligned}$$

- $q_{\tau_{\mathcal{S}}}(term, qualifier) = \{id_i \mid \forall\ i\ \exists\ element_{i,j} \in schema_j$:

$$\begin{aligned} getId(element_{i,j}) &= id_i \\ getTerm(element_{i,j}) &= term \\ getQualifier(element_{i,j}) &= qualifier\} \end{aligned}$$

⌟

To build the thesaurus $\tau_{\mathcal{S}}$, we use the procedure `buildThesaurus` of Algorithm 7.1 on the set of schemas $\mathcal{S}$ obtained through the Adapt phase.

Algorithm 7.1 BuildThesaurus : Build the thesaurus of a given set of schemas

Require: $\mathcal{S} = \{schema_1, ..., schema_n\} \wedge \ \forall i \in [1, n] : \exists entityType_i \in schema_i$
Ensure: $\tau_{\mathcal{S}}$ is a thesaurus for $\mathcal{S}$

```
1: procedure BUILDTHESAURUS(S, τ_S)
2:     reset(τ_S)
3:     for all schema_i ∈ S do
4:         for all entityType_{i_j} ∈ schema_i do
5:             addEntryToThesaurus(τ_S, entityType_{i_j})   ▷ See Algorithm 7.2 on page 98
6:         end for
7:     end for
8: end procedure
```

Building the subsets of semantically similar terms

The thesaurus $\tau_{\mathcal{S}}$ holds the set $\mathscr{T}$ of terms used in the user-drawn interfaces, which we want to partition into subsets of semantically similar terms. To build $\mathcal{T}_{\mathscr{T}}$, the set of semantically similar subsets of $\mathscr{T}$, we apply Algorithm 7.3 on $\tau_{\mathcal{S}}$, using the string distance metric sdm, the threshold t_{sdm}, the ontology $\mathcal{O}$ and the synonyms function $s_{\mathcal{O}}$.

Applying this algorithm on the schemas of Fig. 6.3 would typically yield the following relevant (i.e. containing more than one term) semantically similar subsets :

- $\mathcal{T}_1$ = {“Code”, “Zip code”}
- $\mathcal{T}_2$ = {“Customer”, “Customer number”, “Vat number”, “Number”}
- $\mathcal{T}_3$ = {“Date”, “Hourly rate”}
- $\mathcal{T}_4$ = {“Name”, “First name”, “Last name”}
- $\mathcal{T}_5$ = {“Order”, “Orders”}
- $\mathcal{T}_6$ = {“Product”, “Products”}
- $\mathcal{T}_7$ = {“Provider (primary)”, “Supplier (secondary)”}

7.1.3 Submitting terminological ambiguities to end-users for arbitration

Defining the subsets of semantically equivalent elements

Once the set $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, ...\}$ has been built from $\tau_{\mathcal{S}}$, we can use the latter to map any term $t_{i_j} \in \mathcal{T}_i$ to its corresponding data elements, and furthermore, to their interface widget counterparts. We can therefore visually point out the

Algorithm 7.2 AddEntryToThesaurus : Add an entry to a given thesaurus

Require: $\emptyset$

Ensure: τ contains the mappings l_τ and q_τ for the new element and its descendants, if any.

```
1: procedure ADDENTRYTOTHESAURUS(τ, element)
2:     id ← getId(element)
3:     label ← getLabel(element)
4:     term ← getTerm(element)
5:     qualifier ← getQualifier(element)
6:     ▷ Get the existing mappings for the labels
7:     λ ← l_τ(term)  ▷ λ = {(label_i, {id_ij})}
8:     ι ← l_τ(term, label)  ▷ ι = {id_i}
9:     ▷ Update the mapped sets
10:    λ ← λ\{(label, ι)}
11:    if ι = ∅ then
12:        ι = {id}
13:    else
14:        ι = ι ∪ {id}
15:    end if
16:    λ ← λ ∪ {(label, ι)}
17:    ▷ Update the mappings functions of the thesaurus
18:    ASK> define: l_τ(term, label) ↦ ι
19:    ASK> define: l_τ(term) ↦ λ
20:    ▷ Get the existing mappings for the qualifiers
21:    λ ← q_τ(term)  ▷ λ = {(qualifier_i, {id_ij})}
22:    ι ← q_τ(term, qualifier)  ▷ ι = {id_i}
23:    ▷ Update the mapped sets
24:    λ ← λ\{(qualifier, ι)}
25:    if ι = ∅ then
26:        ι = {id}
27:    else
28:        ι = ι ∪ {id}
29:    end if
30:    λ ← λ ∪ {(qualifier, ι)}
31:    ▷ Update the mappings functions of the thesaurus
32:    ASK> define: q_τ(term, qualifier) ↦ ι
33:    ASK> define: q_τ(term) ↦ lambda
34:    if getType(element) ∈ {ENTITYTYPE, COMPOUNDATTRIBUTE} then
35:        children ← getAttributes(element)
36:        for all child_i ∈ children do
37:            addEntryToThesaurus(τ, child_i)
38:        end for
39:    end if
40: end procedure
```

Algorithm 7.3 BuildSemanticallySimilarSubsets : Build the set $\mathcal{T}$ of semantically similar subsets for a given thesaurus τ

Require: τ, sdm, t_{sdm}, $\mathcal{O}$ and $s_{\mathcal{O}}$
Ensure: $\mathcal{T}$ contains the set of semantically similar subsets of τ with respect to sdm, t_{sdm}, $\mathcal{O}$ and $s_{\mathcal{O}}$

1: **procedure** BuildSemanticallySimilarSubsets($\mathcal{T}$,τ,sdm, t_{sdm}, $\mathcal{O}$, $s_{\mathcal{O}}$)
2: $\mathscr{T} \leftarrow \{terms_i \mid \forall i \in [0,n] : terms_i \in \tau\}$
3: $\mathcal{T} \leftarrow \emptyset$
4: **for** $i = 1$ to n **do** ▷ is there already a subset containing the current term?
5: **if** $\exists\, \mathcal{T}_k \in \mathcal{T} : t_i \in \mathcal{T}_k$ **then**
6: $\mathcal{T}_i \leftarrow \mathcal{T}_k$
7: **else**
8: $\mathcal{T}_i \leftarrow \{t_i\}$
9: $\mathcal{T} \leftarrow \mathcal{T} \cup \{\mathcal{T}_i\}$
10: **end if**
11: **for** $j = i + 1$ to n **do** ▷ for all the remaining terms...
12: **if** $\exists\, \mathcal{T}_l \in \mathcal{T} : t_j \in \mathcal{T}_l$ **then**
13: $\mathcal{T}_j \leftarrow \mathcal{T}_l$
14: **else**
15: $\mathcal{T}_j \leftarrow \{t_j\}$
16: **end if**
17: $\mathcal{T} \leftarrow \mathcal{T} \backslash \mathcal{T}_i$
18: $\mathcal{T} \leftarrow \mathcal{T} \backslash \mathcal{T}_j$
19: **if** $t_i \overset{sdm,t_{sdm},\mathcal{O},s_{\mathcal{O}}}{\Longleftrightarrow} t_j$ **then** ▷ t_i and t_j are semantically similar
20: $\mathcal{T}_i \leftarrow \mathcal{T}_i \cup \mathcal{T}_j$
21: $\mathcal{T} \leftarrow \mathcal{T} \cup \{\mathcal{T}_i\}$
22: **else**
23: $\mathcal{T} \leftarrow \mathcal{T} \cup \{\mathcal{T}_i\} \cup \{\mathcal{T}_j\}$
24: **end if**
25: **end for**
26: **end for**
27: **end procedure**

discovered similarities between concepts in the user-drawn interfaces, in order to ask end-users to validate or reject them.

Indeed, each set of terms $\mathcal{T}_i \in \mathcal{T}_{\mathscr{T}}$ has a corresponding set of logical elements $\mathcal{E}_{\mathcal{S}_i} \subseteq \mathcal{E}_{\mathcal{S}}$, which can be obtained using Algorithm 7.4. For our running example, we can for instance highlight the elements associated with $\mathcal{T}_1$ as illustrated in Fig. 7.2.

This task consists in deciding which *semantic similarities* are actually genuine *semantic equivalences*, which we can define as follows.

Algorithm 7.4 GetSemanticallySimilarDataElements : Get the data elements associated with a given a term contained in the given set of semantically similar terms

Require: $\emptyset$

Ensure: $\mathcal{E}_{\mathcal{S}_i}$ contains the set of logical elements associated with a given set of terms

1: **procedure** GETSEMANTICALLYSIMILARDATAELEMENTS($\mathcal{T}_i, \mathcal{E}_{\mathcal{S}_i}$)
2: $\quad \mathcal{E}_{\mathcal{S}_i} \leftarrow \{e_{i_j} \in \mathcal{E}_{\mathcal{S}} |\ \text{getTerm}(e_{i_j}) \in \mathcal{T}_i\}$
3: **end procedure**

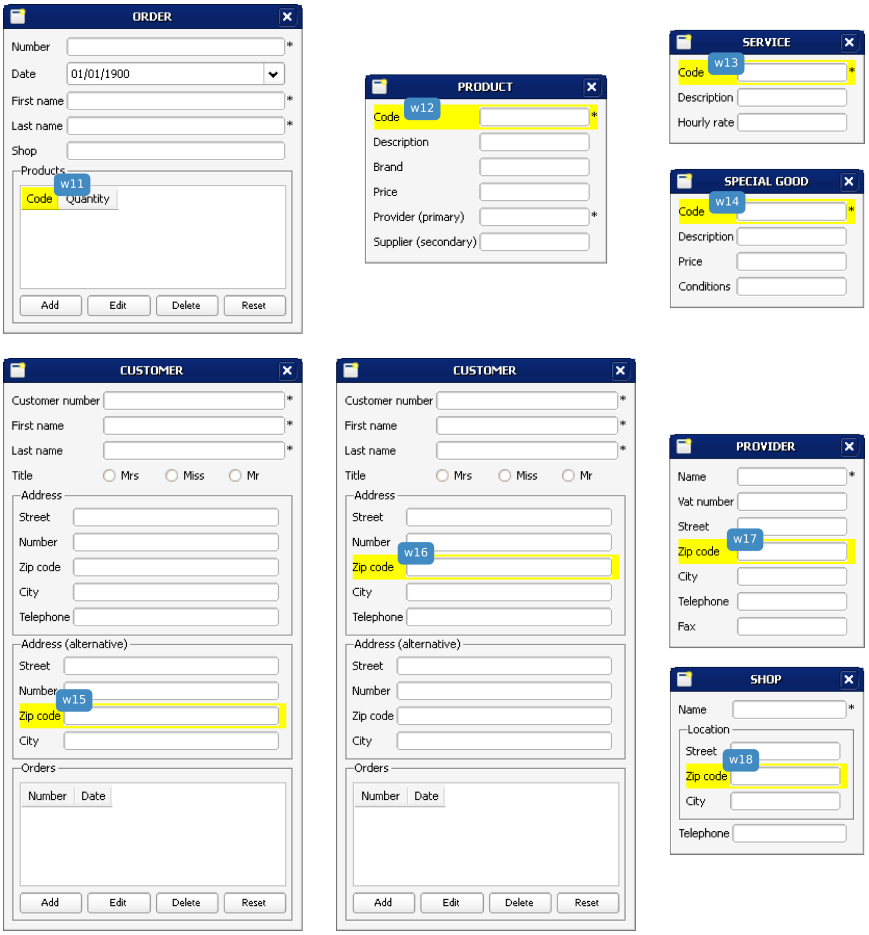


Figure 7.2: Illustration of the set $\mathcal{E}_{\mathcal{S}_1}$ of the semantically similar elements, which are associated with $\mathcal{T}_1$ = {"Code", "Zip code"} for the running example.

Definition 7.6. Two widgets w_1 and w_2 (and by extension, their logical counterpart elements e_1 and e_2) are said to be *semantically equivalent* (noted $w_1 \equiv w_2$ and $e_1 \equiv e_2$ respectively) when it is agreed by the end-users and the analysts that they represent the same concept. ⌟

The validation therefore first consists in examining each $\mathcal{T}_i$ and its associated $\mathcal{E}_{\mathcal{S}_i}$ in order to define the subsets of *semantically equivalent* data elements. More formally, we want to define $\mathcal{E}'_{\mathcal{S}_i} = \{\mathcal{E}'_{\mathcal{S}_{i_1}}, \mathcal{E}'_{\mathcal{S}_{i_2}}, ...\}$ so that :

$$\begin{aligned} \forall i,j,k : \mathcal{E}'_{\mathcal{S}_{i_j}} \cap \mathcal{E}'_{\mathcal{S}_{i_k}} &= \emptyset \\ \forall i \quad : \bigcup_j \mathcal{E}'_{\mathcal{S}_{i_j}} &= \mathcal{E}_{\mathcal{S}_i} \\ \forall i,j,k : e_{i_j}, e_{i_k} \in \mathcal{E}'_{\mathcal{S}_i} \Leftrightarrow e_{i_j} &\equiv e_{i_k} \end{aligned} \tag{7.1}$$

The widgets associated with $\mathcal{T}_1 = \{$"Code", "Zip code"$\}$ in Fig. 7.2 could for instance be grouped into:

- $\mathcal{E}_{\mathcal{S}_{1_1}} = \{e_{1_1}, e_{1_2}, e_{1_3}, e_{1_4}\}$ (corresponding to the widgets $\{w_{1_1}, w_{1_2}, w_{1_3}, w_{1_4}\}$)
- $\mathcal{E}_{\mathcal{S}_{1_2}} = \{e_{1_5}, e_{1_6}, e_{1_7}, e_{1_8}\}$ (corresponding to the widgets $\{w_{1_5}, w_{1_6}, w_{1_7}, w_{1_8}\}$)

This would illustrate that the widgets of $\mathcal{E}_{\mathcal{S}_{1_1}}$ represent a same concept different from the one of the widgets of $\mathcal{E}_{\mathcal{S}_{1_2}}$. Coincidentally, the widgets of $\mathcal{E}_{\mathcal{S}_{1_1}}$ and $\mathcal{E}_{\mathcal{S}_{1_2}}$ each bear the same terms, respectively "Code" and "Zip code". However, this is not necessarily always the case.

Consider for instance the subset $\mathcal{T}_6 = \{$"Product", "Products"$\}$ and its associated elements illustrated in Fig. 7.3. The arbitration could here lead to a single subset $\mathcal{E}_{\mathcal{S}_{6_1}} = \{e_{6_1}, e_{6_2}\}$ of semantically equivalent elements bearing different labels.

Defining unifying terms for semantically equivalent elements

Consequently, since each of these $\mathcal{E}'_{\mathcal{S}_{i_j}}$ may be associated to several non exclusive terms, we want to unify the terminology of these *semantically equivalent* subsets by define a new unique term for each of them, so that we can partition $\mathcal{E}_{\mathcal{S}}$ into $\Gamma = \{(t'_i, \mathcal{E}'_i)\}$ so that :

$$\begin{aligned} &\forall i,j \ : \ t'_i \neq t'_j \\ &\forall i,j \ : \ \mathcal{E}'_i \cap \mathcal{E}'_j = \emptyset \\ &\bigcup_i \mathcal{E}'_i = \mathcal{E}_{\mathcal{S}} \end{aligned} \tag{7.2}$$

This distinctive term may be a new one or can be chosen among the terms of τ_S. A new qualifier can also be assigned individually to put the elements of

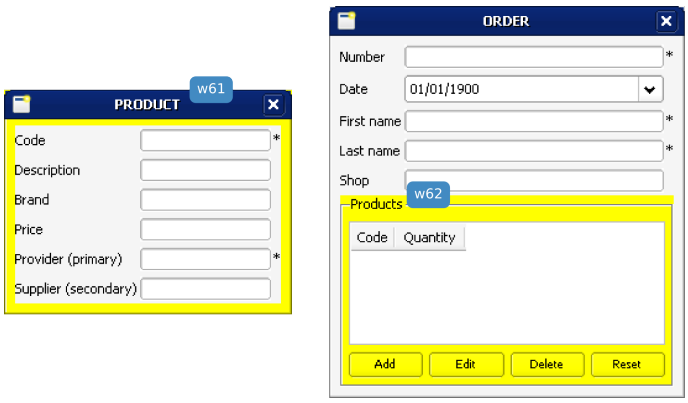


Figure 7.3: Illustration of the set $\mathcal{E}_{\mathcal{S}_6}$ of the semantically similar elements, which are associated with $\mathcal{T}_6 = \{$"Product", "Products"$\}$ for the running example.

a same $\mathcal{E}'_{\mathcal{S}_{i_j}}$ back into context. The labels may also be updated to reflect the new terms, but according to the circumstances (for instance, to indicate the plurality), it might be preferable to keep the same label.

For our example, this could typically include unifying the terminology of the widgets initially associated with the following terms:

- `Order` and `Orders` ($\mathcal{T}_5$) into `Order`, while keeping the label of `Orders` unchanged;
- `Product` and `Products` ($\mathcal{T}_6$) into `Product`, while keeping the label of `Products` unchanged;
- `Provider` and `Supplier` ($\mathcal{T}_7$) into `Provider`, while propagating the modification to the labels.

7.1.4 Processing the terminological decisions of the end-users

The final step of this update consists in updating the widgets, the schemas and the mappings of thesaurus τ_S based on the new terminology $\Gamma = \{(t'_i, \mathcal{E}'_i)\}$.

Each previously defined set $\mathcal{E}'_{\mathcal{S}_{i_j}}$ may involve logical elements of different natures, typically entity types and simple attributes. Given two non necessarily distinct entity types e_1 and e_2, saying that a simple attribute a_{1_i} of e_1 is semantically similar to e_2 implies that a_{1_i} is actually a reference to the entity type e_2. This is typically the case for the term "Provider", which is associated with a form and two fieldsets, as illustrated in Fig. 5.10.

We therefore need to update the form w_1 associated to e_1, as well as its underlying data model to acknowledge this information. This implies removing

the widget w_{1_i} associated with a_{1_i} from w_i, replacing it with a container $w_{1_i}^{'}$ associated with a same term, then inserting w_{1_i} into $w_{1_i}^{'}$ after giving it a new term, which ideally should be the term of one of the widgets of the form w_2 associated to e_2. This procedure can be formalised into Algorithm 7.5.

Algorithm 7.5 TransformReferentialElement : Transform a referential element into an entity type

Require: getType(e_1 = `ATTRIBUTE`)∧getType(e_2 = `ENTITYTYPE`)
∧∃w_1, w_2 corresponding to e_1, e_2
Ensure: ∅

1: **procedure** TRANSFORMREFERENTIALELEMENT(e_1, e_2, w_1, w_2)
2: $w_{1_p} \leftarrow$ getParent(w_1)
3: removeChild(w_{1_p}, w_1)
4: $w_1^{'} \leftarrow$ `null`
5: **while** $w_1^{'}$ = `null` ∨ getType($w_1^{'} \notin$ [`FIELDSET`,`TABLE`]) **do**
6: ASK> define: ($w_1^{'}$) ▷ define a new container and its properties
7: setTerm($w_1^{'}, getTerm(w_2)$)
8: setLabel($w_1^{'}, getLabel(w_2)$)
9: setQualifier($w_1^{'}, getQualifier(w_2)$)
10: **end while**
11: **while** $\exists w \in getDirectChildren(w_{1_p}) : ((getTerm(w) = getTerm(w_1^{'}) \wedge getQualifier(w) = getQualifier(w_1^{'})) \vee (getLabel(w) = getLabel(w_1^{'}) \wedge getQualifier(w) = getQualifier(w_1^{'})))$ **do**
12: ASK> define: ($q_1^{'}$) ▷ define an alternative qualifier to prevent widgets with the same combinations
13: setQualifier($w_1^{'}, q_1^{'}$)
14: **end while**
15: ASK> define: (t_1, l_1, q_1) ▷ define the new properties for the original referential widget
16: setTerm(w_1, t_1)
17: setLabel(w_1, l_1)
18: setQualifier(w_1, q_1)
19: addChild($w_1^{'}, w_1$)
20: addChild($w_{1_p}, w_1^{'}$)
21: UNFOLD(w_{1_p}) ▷ see Algorithm 6.5 on page 88
22: **end procedure**

In our case, this would imply replacing the inputs `Provider (primary)` and `Provider (secondary)` of the form `Product` by two fieldsets, each containing an input named `Name`.

The whole validation process can hence be described by Algorithm 7.6, and as can be observed, it implies that all the *semantically equivalent* elements of the schemas now bear the same term, and vice-versa. Moreover, logical elements

can now only be semantically equivalent to elements of the same nature (simple attributes with simple attributes and entity types with entity types).

Fig. 7.4 illustrates the update of the forms, while Fig. 7.5 illustrates the update of the schemas for our running example.

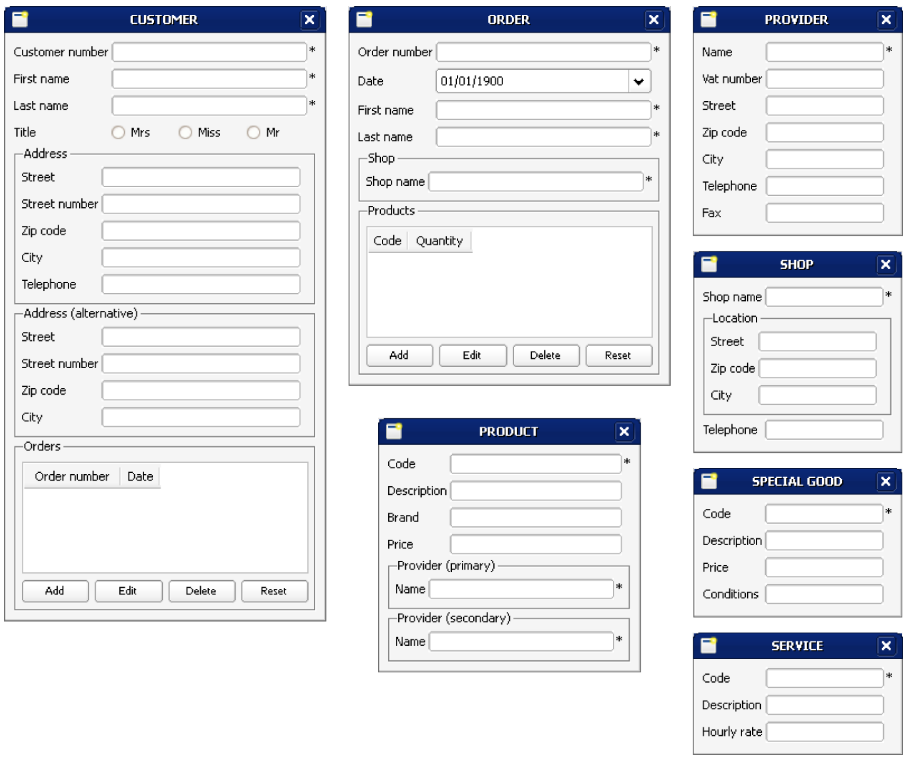


Figure 7.4: The updated forms of the example after the validation of the semantic redundancies.

7.1.5 Choosing appropriate String Metrics

The principles of our approach are generic and need to be instantiated using appropriate Strings Metrics and Ontologies.

Among available String metrics, we choose to work around Jaro-Winkler's metrics [Winkler, 1990], which has proven to be a good fit for short strings. Jaro-Winkler's metrics uses a prefix scale p which gives more favourable ratings to strings that match from the beginning for a set prefix length ℓ. Given two strings l_1 and l_2, their Jaro-Winkler's similarity index $d_{jw_{1,2}}$ is calculated as

Algorithm 7.6 ValidateSemanticSimilarities : Validate the semantically similar subsets of a given thesaurus

Require: $\emptyset$
Ensure: $\emptyset$

procedure VALIDATESEMANTICSIMILARITIES(τ,sdm, t_{sdm}, $\mathcal{O}$, $s_{\mathcal{O}}$)
  BUILDTHESAURUS($\mathcal{S}, \tau$)
  BUILDSEMANTICALLYSIMILARSUBSETS($\mathcal{T}, \tau, sdm, t_{sdm}, \mathcal{O}, s_{\mathcal{O}}$)
  for all $\mathcal{T}_i \in \mathcal{T}$ **do**
    GETSEMANTICALLYSIMILARDATAELEMENTS($\mathcal{T}_i, \mathcal{E}_{\mathcal{S}_i}$)
    ASK> define: ($\mathcal{E}^{'}_{\mathcal{S}_i}$) ▷ define the semantically equivalent subsets based on $\mathcal{T}_i$ and $\mathcal{E}_{\mathcal{S}_i}$
  end for
  $\mathscr{T}^{'} \leftarrow \emptyset$
  $\Gamma \leftarrow \emptyset$
  for all $\mathcal{E}^{'}_{\mathcal{S}_{i_j}} \in \bigcup_i \{\mathcal{E}^{'}_{\mathcal{S}_i}\}$ **do**
    $t^{'}_{i_j} \leftarrow$ `null`
    while $t^{'}_{i_j} =$ `null` $\vee$ $t^{'}_{i_j} \in \mathscr{T}^{'}$ **do**
      ASK> define: ($t^{'}_{i_j}$) ▷ define a unique term
    end while
    for all $e_{i_{j_k}} \in \mathcal{E}^{'}_{\mathcal{S}_{i_j}}$ **do**
      setMetaproperty($e_{i_{j_k}}, t^{'}_{i_j}$, `term`)
      setTerm($w_{i_{j_k}}, t^{'}_{i_j}$)
      if relevant **then**
        ASK> define: ($l^{'}_{i_j}$)
        setMetaproperty($e_{i_{j_k}}, l^{'}_{i_j}$, `label`)
        setLabel($w_{i_{j_k}}, l^{'}_{i_j}$)
      end if
      if necessary **then**
        ASK> define: ($q^{'}_{i_j}$)
        setMetaproperty($e_{i_{j_k}}, q^{'}_{i_j}$, `qualifier`)
        setQualifier($w_{i_{j_k}}, q^{'}_{i_j}$)
      end if
      if getType($e_{i_{j_k}}$ = `ATTRIBUTE`) $\wedge$ ($\exists e_{i_{j_l}} \in \mathcal{E}^{'}_{\mathcal{S}_{i_j}}$|getType($e_{i_{j_l}}$) = `ENTITYTYPE`) **then**
        TRANSFORMREFERENTIALELEMENT($e_{i_{j_k}}, e_{i_{j_l}}, w_{i_{j_k}}, w_{i_{j_l}}$)
      end if
    end for
    $\mathscr{T}^{'} \leftarrow \mathscr{T}^{'} \cup \{t^{'}_{i_j}\}$
    $\Gamma = \Gamma \cup \{(t^{'}_{i_j}, \mathcal{E}^{'}_{\mathcal{S}_{i_j}})\}$
  end for
  BUILDTHESAURUS($\mathcal{S}, \tau$)
end procedure

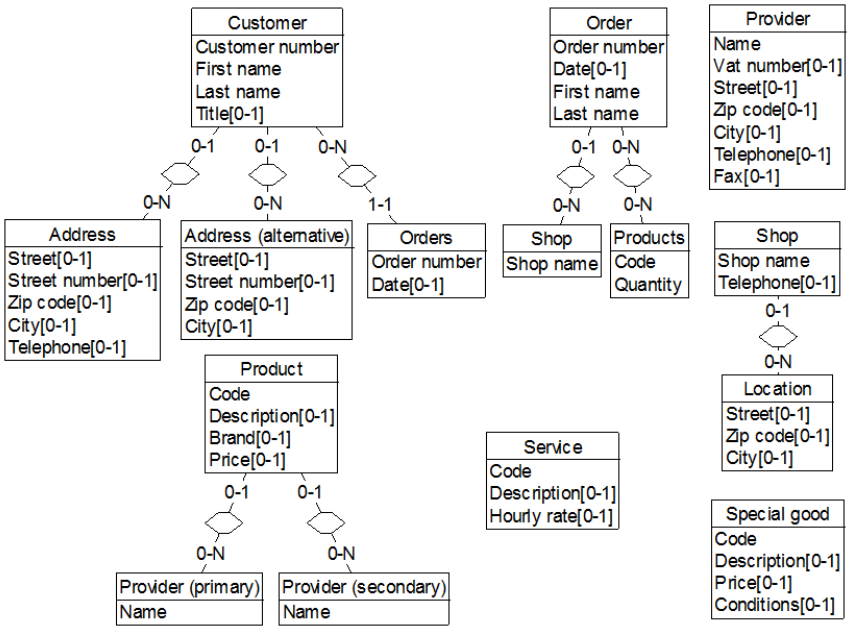


Figure 7.5: The updated schemas of the example after the validation of the semantic redundancies.

follows:

$$d_{jw}(l_1, l_2) = d_j(l_1, l_2) + (\ell p(1 - d_j(l_1, l_2))) \tag{7.3}$$

where:

- $d_{j_{1,2}}$ is the Jaro similarity index for strings l_1 and l_2:

$$d_j(l_1, l_2) = \frac{1}{3}\left(\frac{m_{1,2}}{|l_1|} + \frac{m_{1,2}}{|l_2|} + \frac{m_{1,2} - t_{1,2}}{m_{1,2}}\right) \tag{7.4}$$

  where:

  - $m_{1,2}$ is the number of matching characters between l_1 and l_2;
  - $t_{1,2}$ is the number of transpositions between l_1 and l_2.

- ℓ is the length of common prefix at the start of the string up to a maximum of 4 characters;
- p is a constant scaling factor for how much the score is adjusted upwards for having common prefixes. The standard value for this constant in Winkler's work is $p = 0.1$

Since Jaro-Winkler's metrics is actually a similarity index ranging from 0 (different) to 1 (equal) and that we want to agree with Definition 7.1, we need to adapt this metrics so that its smaller scores correspond to higher similarities. For this purpose, we define the *Jaro-Winkler's inverted similarity index* d_{jwi} for two strings l_1 and l_2 as follows:

$$d_{jwi}(l_1, l_2) = 1 - d_{jw}(l_1, l_2) \tag{7.5}$$

Since the longest common prefix impacts the similarity index, we observe that comparing reversed strings may yield better results in certain cases. For instance, Table 7.1 shows Jaro-Winkler's inverted similarity index applied to the strings "Name", "First Name", "Last Name", "Family Name" and their reversed version. We can notably see that that "Name" and "Last Name" are much closer (0.19) in their reversed versions that in their normal versions (0.55).

Label 1	**Label 2**	d_{jwi}	**Label 1 bis**	**Label 2 bis**	d_{jwi}
Name	Name	0	Eman	Eman	0
Name	First Name	1	Eman	Eman tsrif	0.2
Name	Last Name	0.55	Eman	Eman tsal	0.19
Name	Family Name	0.44	Eman	Eman ylimaf	0.21
First Name	Last Name	0.17	Eman tsrif	Eman tsal	0.17
First Name	Family Name	0.2	Eman tsrif	Eman ylimaf	0.22
Last Name	Family Name	0.25	Eman tsal	Eman ylimaf	0.25

Table 7.1: Jaro-Winkler's inverted similarity index (d_{jwi}) applied to example strings and their reversed version.

In order to take in account these observations, we therefore define the *Jaro-Winkler-based similarity index* d_{jwb} for two labels l_1 and l_2, as follows:

$$d_{jwb}(l_1, l_2) = Minimum(d_{jwi}(l_1, l_2), d'_{jwi}(l_1, l_2)) \tag{7.6}$$

where d'_{jwi} is Jaro-Winkler's inverted similarity index applied to the reversed version of the labels l_1 and l_2.

And since we observed that $t_{jwb} = 0.2$ was a reasonable threshold, we define two labels s_1 and s_2 as *orthographically similar* with respect to Jaro-Winkler-based similarity index, iff $d_{jwb}(s_1, s_2) \leq t_{jwb}$.

7.1.6 Choosing appropriate Ontologies

Besides, to investigate the ontological aspect, and more precisely the synonymy issue, we choose to take advantage of *WordNet*[Fellbaum, 1998], which is not precisely an ontological tool, but nevertheless an English non domain-specific orthographical reference system, handling nouns, verbs, adjectives and adverbs, and providing definitions, synonyms and hypernyms (i.e. generalisation of words).

7.1.7 Reducing terminological redundancies

In order to reduce the semantic redundancies upstream, i.e. during the drawing phase, we can take advantage of our definition of semantic similarity to provide an on-the-fly terminology suggester and analyser. When inserting a new widget or editing an existing one, the suggester automatically proposes possible terms, labels and qualifiers based on the existing terminology. If the user chooses to provide his own term, label and qualifier, the analyser compares them to the existing terminology to detect possible similarities and ask the user for direct arbitration. This can help to reduce typing mistakes and the use of synonyms in order to unify the terminology from the start.

7.2 Structural ambiguities

The second type of similarity that may occur is the *structural similarity*. Typically, we can observe that interface containers (i.e. forms, fieldsets or tables) bearing different labels may contain interface elements who share semantically similar labels. This is for instance the case for:

- the fieldsets `Address`, the fieldset `Location` and the form `Provider`, who all share at least `Street`, `Zip Code` and `City` (Fig. 7.6);
- the forms `Customer` and `Order`, who share `First Name` and `Last Name` (Fig. 7.7);
- the forms `Product`, `Special good` and `Service`, who all share at least `Code` and `Description` (Fig. 7.8).

Intuitively, given the chain of transformations that we went through, one may sense that at the logical level, such redundancies involve entity types associated to interface containers (i.e. interfaces, fieldsets and tables) and having semantically similar attributes, as well as relationships with other semantically similar entity types (Fig. 7.9). This implies looking for *patterns* across the schemas of $\mathcal{S}$, which could lead to merging or connecting different concepts.

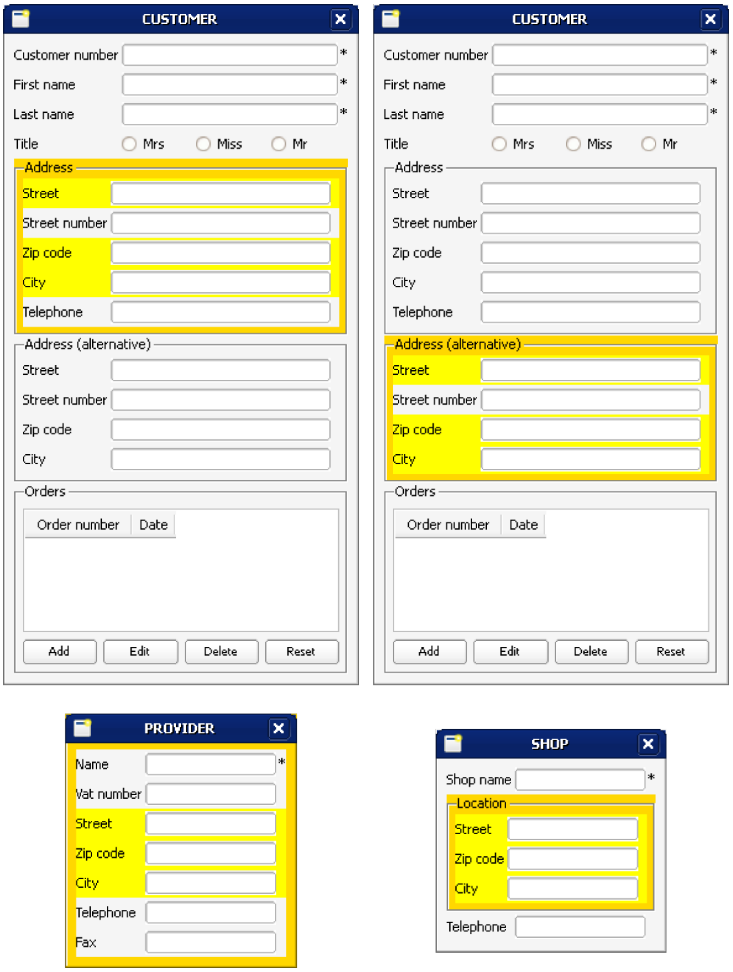


Figure 7.6: The fieldsets `Address`, the fieldset `Location` and the form `Provider`, who all share at least the widgets `Street`, `Zip Code` and `City` in our running example.

7.2.1 Formalising the notion of structural similarity

Most common cases of structural similarity

Intuitively, two entity types are said to be *structurally similar* if they have attributes bearing the same name and/or roles in relationship types involving the same target entity types. Consider for instance the two entity types E_1 and E_2 of Fig. 7.10(a), which share the attributes A and B (the following principles are the same for shared roles). Let us recall that, as mentioned in

Section 3.4.1, the signification of their structural similarity can be classified among the following most common cases:

- *equality* : The two entity types represent the same concept, but went undetected during the semantic analysis. This is typically the case of `Address` and `Location`. Such entity types should be merged into a single concept (Fig. 7.10(b)).
- *specialisation* : One of the two elements is a specialisation of the other (Fig. 7.10(d)), such as a `Seasonal good` that can be seen as a specialised `Product`.
- *union* : The two entity types partially represent the same concept, and could be seen as specialising a higher concept non explicitly expressed (Fig. 7.10(c)). For instance, one could argue that a `Product` and a `Service` are specialisations of the concept of `Solution`.
- *complementarity* : One of the two entity types actually refers to the other (Fig. 7.10(e)). This is typically the case of `Order` which refers to `Customer`, or `Provider` which refers to `Address`.
- *difference* : Finally, two entity types can also fortuitously share a same of attributes, while being intrinsically different. For instance, a `Subcontractor` and a `Supplier` may share properties such as `Name` and `Address`, but they represent different concepts.

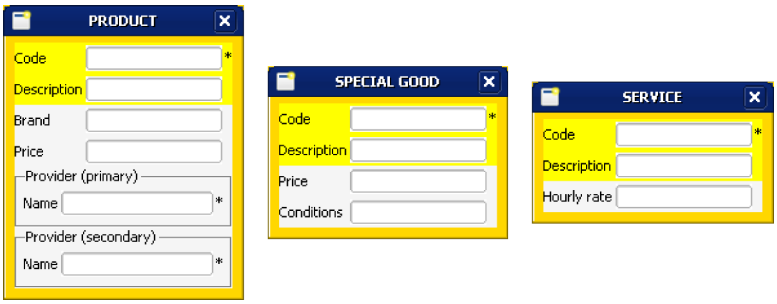


Figure 7.7: The forms `Product`, `Special good` and `Service`, who all share at least the widgets `Code` and `Description` in our running example.

Attribute and role pattern for two entity types

As we have seen, the structural similarity of two entity types actually relies on the *semantically equivalent* attributes that they have in common, as well as the *semantically equivalent* entity types with which they have a relationship.

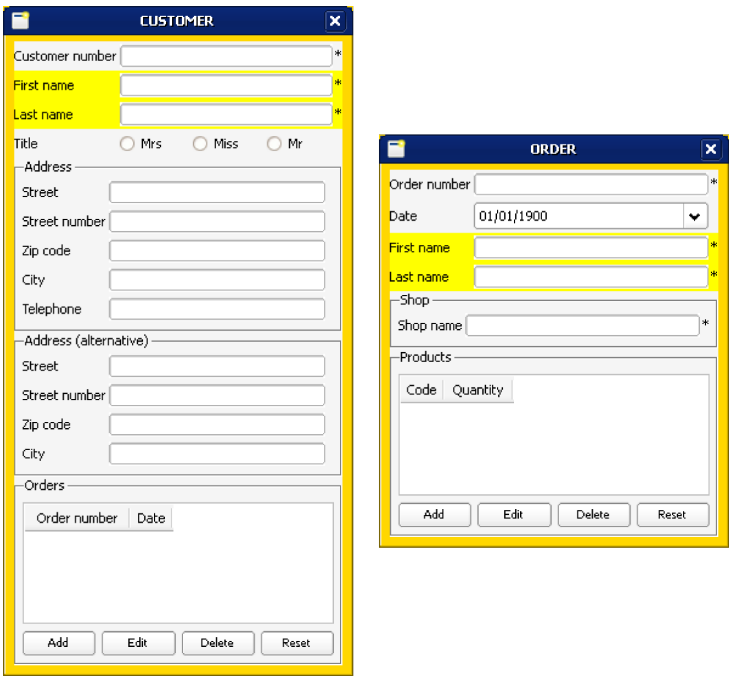


Figure 7.8: The forms `Customer` and `Order`, who share the widgets `First Name` and `Last Name` in our running example.

Regarding the attributes, a *pattern* can be defined as a bijection between two sets of attributes belonging to a different entity type. More formally:

Definition 7.7. Given two entity types e_1 and e_2, and their associated sets of attributes $\mathcal{A}_1$ and $\mathcal{A}_2$, $p = \{(\mathcal{A}_{1_1}, \mathcal{A}_{2_1}), ..., (\mathcal{A}_{1_n}, \mathcal{A}_{2_n})\}$ is an *attribute pattern* of n for e_1 and e_2 iff:

$$
\begin{aligned}
&\forall i \in [1,2], j \in [1,n] && : \mathcal{A}_{i_j} \subseteq \mathcal{A}_i \\
&\forall i \in [1,2], j,k \in [1,n] && : \mathcal{A}_{i_j} \cap \mathcal{A}_{i_k} = \emptyset \\
&\forall i \in [1,2], j \in [1,n] && : a_{i_{j_k}}, a_{i_{j_l}} \in \mathcal{A}_{i_j} \Rightarrow a_{i_{j_k}} \equiv a_{i_{j_l}} \\
&\forall i \in [1,2], j \in [1,n] && : a_{i_{j_k}} \in \mathcal{A}_{i_j} \wedge a_l \in \mathcal{A}_i \backslash \mathcal{A}_{i_j} \Rightarrow a_{i_{j_k}} \not\equiv a_l \\
&\forall j \in [1,n] && : a_{1_{j_k}} \in \mathcal{A}_{1_j} \wedge a_{2_{j_l}} \in \mathcal{A}_{2_j} \Rightarrow a_{1_{j_k}} \equiv a_{2_{j_l}}
\end{aligned}
$$

⌟

Until now, we only considered the semantic equivalence of entity types and attributes. Let us now define it for roles as well.

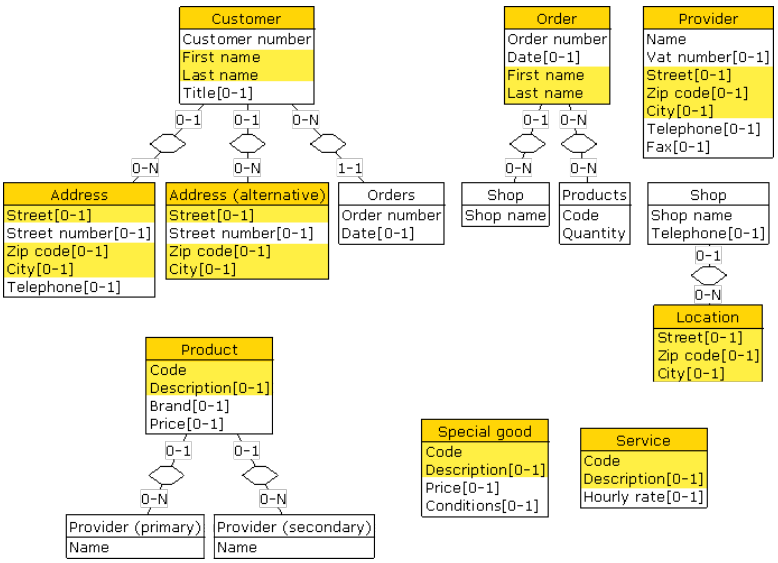


Figure 7.9: The structural redundancies within the schemas corresponding to the forms illustrated in Fig. 7.6, Fig. 7.7 and Fig. 7.8.

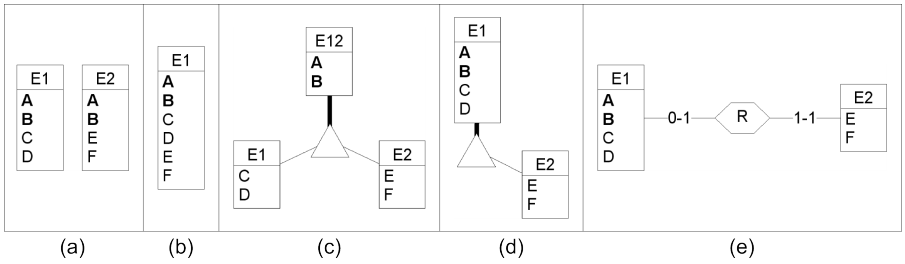


Figure 7.10: Typical cases of structural similarity.

Definition 7.8. Two roles r_1 and r_2, respectively played by entity types e_1 and $e2$, are said to be *semantically equivalent* (noted $r_1 \equiv r_2$) iff $e_{r_1} \equiv e_{r_2}$, with e_{r_i} being the entity type associated to e_i through the relationship involving the role r_i ⌟

In the scope of our research, we will actually consider that:

$$e_{r_i} = getMetaproperty(r_i, \texttt{targetEntityType})$$

in order to focus on the logical counterparts of referential widgets. We can therefore have $e_{r_i} = \texttt{null}$ if that metaproperty has not been set for r_i.

As for the attributes, a *role pattern* can therefore be defined as a bijection between sets of semantically equivalent roles being played by different entity types. More formally:

Definition 7.9. Given two entity types e_1 and e_2, and their associated sets of roles $\mathcal{R}_1$ and $\mathcal{R}_2$, $p = \{(\mathcal{R}_{1_1}, \mathcal{R}_{2_1}), ..., (\mathcal{R}_{1_n}, \mathcal{R}_{2_n})\}$ is a *role pattern* of size n for e_1 and e_2 iff:

$$\begin{aligned}
&\forall i \in [1,2], j \in [1,n] && : \mathcal{R}_{i_j} \subseteq \mathcal{R}_i \\
&\forall i \in [1,2], j,k \in [1,n] && : \mathcal{R}_{i_j} \cap \mathcal{R}_{i_k} = \emptyset \\
&\forall i \in [1,2], j \in [1,n] && : r_{i_{j_k}}, r_{i_{j_l}} \in \mathcal{R}_{i_j} \Rightarrow r_{i_{j_k}} \equiv r_{i_{j_l}} \\
&\forall i \in [1,2], j \in [1,n] && : r_{i_{j_k}} \in \mathcal{R}_{i_j} \wedge r_l \in \mathcal{R}_i \backslash \mathcal{R}_{i_j} \Rightarrow r_{i_{j_k}} \not\equiv r_l \\
&\forall j \in [1,n] && : r_{1_{j_k}} \in \mathcal{R}_{1_j} \wedge r_{2_{j_l}} \in \mathcal{R}_{2_j} \Rightarrow r_{1_{j_k}} \equiv r_{2_{j_l}}
\end{aligned}$$

⌟

Coincidentally, *semantically equivalent* attributes and entity types can be asserted easily, thanks to the previous terminological investigation. Indeed, thanks to the partition Γ of the elements of $\mathcal{E}_\mathcal{S}$, all the equivalent elements bear the same term.

Consider for instance the schemas depicted in Figure 7.11, with the entity types e_1, e_2, e_3 and e_4, respectively named "Clerk", "Shop Assistant", "Director" and "Sales Representative". The entity types e_1 and e_2 have the sets of attributes $\mathcal{A}_1 = \{a_{1_1}, a_{1_2}, a_{1_3}\}$ and $\mathcal{A}_2 = \{a_{2_1}, a_{2_2}, a_{2_3}\}$, as well as the sets of roles $\mathcal{R}_1 = \{r_{1_1}, r_{1_2}, r_{1_3}, r_{1_4}, r_{1_5}\}$ and $\mathcal{R}_2 = \{r_{2_1}, r_{2_2}, r_{2_3}\}$.

These two entity types therefore share the following attribute and role patterns:

- $p_1 = \{(\{a_{1_1}\}, \{a_{2_1}\})\}$, which involves the term "First Name";
- $p_2 = \{(\{a_{1_2}\}, \{a_{2_2}\})\}$, which involves the term "Last Name";
- $p_3 = \{(\{a_{1_1}\}, \{a_{2_1}\}), (\{a_{1_2}\}, \{a_{2_2}\})\}$, which involves the terms "First Name" and "Last Name";
- $p_4 = \{(\{r_{1_1}\}, \{r_{2_1}\})\}$, which involves the entity type named "Branch";
- $p_5 = \{(\{r_{1_2}\}, \{r_{2_2}\})\}$, which involves the entity type named "Status";
- $p_6 = \{(\{r_{1_1}\}, \{r_{2_1}\}), (\{r_{1_2}\}, \{r_{2_2}\})\}$, which involves the entity types named "Branch" and "Status".

Component pattern and structural similarity for two entity types

Based on the definitions of attribute and role patterns, we can now define the notion of *component pattern* for two entity types.

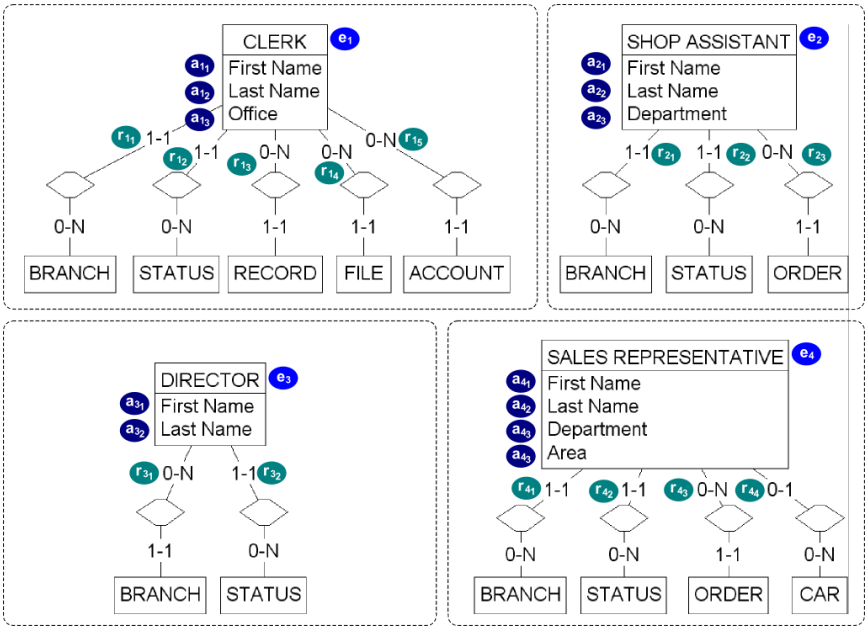


Figure 7.11: A few example schemas illustrating different patterns.

Definition 7.10. Given two entity types e_1 and e_2, their associated sets of components $\mathcal{C}_1 = \mathcal{A}_1 \cup \mathcal{R}_1$ and $\mathcal{C}_2 = \mathcal{A}_2 \cup \mathcal{R}_2$, $p = \{(\mathcal{C}_{1_1}, \mathcal{C}_{2_1}), ..., (\mathcal{C}_{1_n}, \mathcal{C}_{2_n})\}$ is a *component pattern* of size n for e_1 and e_2 iff:

$$\begin{aligned}
&\forall i \in [1,2], j \in [1,n] &&: \mathcal{C}_{i_j} \subseteq \mathcal{C}_i \\
&\forall i \in [1,2], j,k \in [1,n] &&: \mathcal{C}_{i_j} \cap \mathcal{C}_{i_k} = \emptyset \\
&\forall i \in [1,2], j \in [1,n] &&: c_{i_{j_k}}, c_{i_{j_l}} \in \mathcal{C}_{i_j} \Rightarrow c_{i_{j_k}} \equiv c_{i_{j_l}} \\
&\forall i \in [1,2], j \in [1,n] &&: c_{i_{j_k}} \in \mathcal{C}_{i_j} \wedge c_l \in \mathcal{C}_i \backslash \mathcal{C}_{i_j} \Rightarrow c_{i_{j_k}} \not\equiv c_l \\
&\forall j \in [1,n] &&: c_{1_{j_k}} \in \mathcal{C}_{1_j} \wedge c_{2_{j_l}} \in \mathcal{C}_{2_j} \Rightarrow c_{1_{j_k}} \equiv c_{2_{j_l}}
\end{aligned}$$

⌟

We also define the components of a pattern and the pattern components of an entity type as follows.

Definition 7.11. Given a component pattern $p = \{(\mathcal{C}_{1_1}, \mathcal{C}_{2_1}), ..., (\mathcal{C}_{1_n}, \mathcal{C}_{2_n})\}$ for two entity types e_1 and e_2 and their associated sets of components $\mathcal{C}_1$ and $\mathcal{C}_2$, the set of the *components of the pattern* p is defined as:

$$\mathcal{C}^p = \{c \mid \exists i \in [1,2], j \in [1,n] : c \in \mathcal{C}_{i_j}\}$$

⌟

Definition 7.12. Given a component pattern p for two entity types e_i and e_j and their associated sets of components $\mathcal{C}^p$, $\mathcal{C}_i$ and $\mathcal{C}_j$, the set of the *pattern's components for the entity type* e_i is defined as:

$$\mathcal{C}_i^p = \{c \mid c \in \mathcal{C}_i \cap \mathcal{C}^p\}$$

⌟

In the example of Fig. 7.11, the pattern $p_7 = \{(\{a_{2_1}\}, \{a_{4_1}\}), (\{a_{2_2}\}, \{a_{4_2}\}), (\{a_{2_3}\}, \{a_{4_3}\})(\{r_{2_1}\}, \{r_{4_1}\}), (\{r_{2_2}\}, \{r_{4_2}\}), (\{r_{2_3}\}, \{r_{4_3}\})\}$ is a component pattern of size 6 for the entity types e_2 (`Shop assistant`) and e_4 (`Sales representative`).

Subsequently, we can define the *maximal pattern* for two entity types e_1 and e_2 as the largest pattern for these entity types, which is unique since the definition of a component pattern relies on semantic equivalence. More formally:

Definition 7.13. Given two entity types e_1 and e_2, their associated sets of components $\mathcal{C}_1$ and $\mathcal{C}_2$, and $\mathcal{P}_{1,2}$ the set of all the components patterns for e_1 and e_2, $\tilde{\rho}_{1,2} \in \mathcal{P}_{1,2}$ is said to be the *maximal component pattern* for e_1 and e_2 iff:

$$\forall p \neq \tilde{\rho}_{1,2} \in \mathcal{P}_{1,2} : |p| < |\tilde{\rho}_{1,2}|$$

⌟

In our example, p_7 is therefore the maximal component pattern for the entity types e_2 (`Shop assistant`) and e_4 (`Sales representative`).

Intuitively, we will therefore consider two entity types e_1 and e_2 as structurally similar if their component maximal pattern is not null. More formally:

Definition 7.14. Two entity types e_1 and e_2 are said to be *structurally similar* to the degree n (noted $e_1 \overset{n}{\rightsquigarrow} e_2$), iff $|\tilde{\rho}_{1,2}| = n$. ⌟

Structural similarity for a set of entity types

By extension, we can also define the notion of a pattern for a set of entity types, which is basically a pattern that occurs between any pair of these entity types. More formally:

Definition 7.15. Given a set of entity types $\mathfrak{E} = \{e_1, ..., e_m\}$, and their associated sets of components $\mathcal{C}_1, ..., \mathcal{C}_m$, $p = \{(\mathcal{C}_{1_1}, ..., \mathcal{C}_{m_1}), ..., (\mathcal{C}_{1_n}, ..., \mathcal{C}_{m_n})\}$ is a *component pattern* of size n for $\mathfrak{E}$ iff:

$$\begin{aligned}
&\forall i \in [1, m], j \in [1, n] &&: \mathcal{C}_{i_j} \subseteq \mathcal{C}_i \\
&\forall i \in [1, m], j, k \in [1, n] &&: \mathcal{C}_{i_j} \cap \mathcal{C}_{i_k} = \emptyset \\
&\forall i, j \in [1, m] &&: p = \{(\mathcal{C}_{i_1}, \mathcal{C}_{j_1}), ..., (\mathcal{C}_{i_n}, \mathcal{C}_{j_n})\} \in \mathcal{P}_{i,j}
\end{aligned}$$

⌟

Similarly to pairs of entity types, we also define the components of a pattern and the pattern components of an entity type as follow.

Definition 7.16. Given a component pattern $p = \{(\mathcal{C}_{1_1}, ..., \mathcal{C}_{m_1}), ..., (\mathcal{C}_{1_n}, ..., \mathcal{C}_{m_n})\}$ for a set of entity types $\mathfrak{E} = \{e_1, ..., e_m\}$ and their associated sets of components $\mathcal{C}_1, ..., \mathcal{C}_m$, the set of the *components of the pattern* p is defined as:

$$\mathcal{C}^p = \{c \mid \exists i \in [1, m], j \in [1, n] : c \in \mathcal{C}_{i_j}\}$$

⌟

Definition 7.17. Given a component pattern p (with the components $\mathcal{C}^p$) for a set of entity types $\mathfrak{E}$ containing e_i (with the components $\mathcal{C}_i$), the set of the *pattern's components for the entity type* e_i is defined as:

$$\mathcal{C}_i^p = \{c \mid c \in \mathcal{C}_i \cap \mathcal{C}^p\}$$

⌟

In our example, $p = \{(\{a_{1_1}\}, \{a_{2_1}\}, \{a_{3_1}\}, \{a_{4_1}\}), (\{a_{1_2}\}, \{a_{2_2}\}, \{a_{3_2}\}, \{a_{4_2}\}), (\{r_{1_1}\}, \{r_{2_1}\}, \{r_{3_1}\}, \{r_{4_1}\})\}$, which is based on the attributes named "First Name" and "Last Name" and the roles involved with the entity types named "Branch", is a component pattern of size 3 for $\mathfrak{E} = \{e_1, e_2, e_3, e_4\}$.

Subsequently, we can also define the *maximal pattern* for a set of entity types $\mathfrak{E}$, as the largest pattern for this set of entity types. More formally:

Definition 7.18. Given a set of entity types $\mathfrak{E}$, and $\mathcal{P}_\mathfrak{E}$ the set of all the component patterns for $\mathfrak{E}$, $\tilde{\rho}_\mathfrak{E} \in \mathcal{P}_\mathfrak{E}$ is said to be the *maximal component pattern* for $\mathfrak{E}$ iff:

$$\forall p \neq \tilde{\rho}_\mathfrak{E} \in \mathcal{P}_\mathfrak{E} : |p| < |\tilde{\rho}_{\mathfrak{E}|}$$

⌟

In our example, $p = \{(\{a_{1_1}\}, \{a_{2_1}\}, \{a_{3_1}\}, \{a_{4_1}\}), (\{a_{1_2}\}, \{a_{2_2}\}, \{a_{3_2}\}, \{a_{4_2}\}), (\{r_{1_1}\}, \{r_{2_1}\}, \{r_{3_1}\}, \{r_{4_1}\}), (\{r_{1_2}\}, \{r_{2_2}\}, \{r_{3_2}\}, \{r_{4_2}\})\}$, which is based on the attributes named "First Name" and "Last Name" and the roles involved with the entity types named "Branch" and "Status", is the maximal component pattern for $\mathfrak{E} = \{e_1, e_2, e_3, e_4\}$.

The definition of structural similarity for a set of entity types leads in turn to the definition of *structurally similar subsets* within a given set of entity types. In such a subset $\mathscr{E}_i$, all the entity types are structurally close enough from at least another entity type of the set, while being structurally different from any other entity type. More formally:

Definition 7.19. Given a set of entity types $\mathfrak{E}$, $\mathscr{E}_i = \{e_{i_1}, ..., e_{i_m}\} \subseteq \mathfrak{E}$ is a *subset of structurally similar entity types* of $\mathfrak{E}$, iff:

$$\begin{aligned}(m = 1 \ \vee \ \forall\, e_{i_j} \in \mathscr{E}_i,\ \exists\, e_{i_k} \neq e_{i_j} \in \mathscr{E}_i &: e_{i_j} \overset{n}{\rightsquigarrow} e_{i_k} \wedge n > 0)\\ \wedge(\forall\, e_{i_j} \in \mathscr{E}_i,\ \nexists\, e_k \in \mathfrak{E} \backslash \mathscr{E}_i &: e_{i_j} \overset{n}{\rightsquigarrow} e_k \wedge n > 0)\end{aligned}$$

⌟

The set $\mathscr{E}_\mathfrak{E}$ that contains all the $\mathscr{E}_i$ forms a partition of $\mathfrak{E}$, so that:

- $\forall i \neq j : \mathscr{E}_i \cap \mathscr{E}_j = \emptyset$
- $\bigcup_i \mathscr{E}_i = \mathfrak{E}$

Consider for instance the set $\mathfrak{E}$ containing the entity types highlighted in Fig. 7.9. The following sets $\mathscr{E}_1$, $\mathscr{E}_2$ and $\mathscr{E}_3$ form a partition of $\mathfrak{E}$:

- $\mathscr{E}_1 = \{$ `Customer`, `Order` $\}$
- $\mathscr{E}_2 = \{$ `Address`, `Address (alternative)`, `Location`, `Provider` $\}$
- $\mathscr{E}_3 = \{$ `Product`, `Special good`, `Service` $\}$

7.2.2 Discovering structural ambiguities

In order to build the *subset of structurally similar entity types* of a given set $\mathfrak{E}$ while storing the maximal pattern between each entity types, we need to analyse the structure of these entity types.

As explained in Section 3.4.1, the structure of user-drawn interfaces is usually quite simple, which implies that traditional tree mining algorithms prove inappropriate. Instead of putting in motion such heavy algorithms, we therefore propose to adopt a simpler approach that consists in comparing one by one each entity types in terms of attributes and directly related entity types.

For this purpose, we use Algorithm 7.7 to build the set of maximal patterns $\Phi_\mathfrak{E} = \{\tilde{\rho}_{i,j}\}$ for each pair of e_i, e_j and the set $\mathscr{E}_\mathfrak{E}$ of structurally similar subsets of $\mathfrak{E}$.

Algorithm 7.7 BuildPatternsSet : Mine maximal patterns

Require: $\mathfrak{E} = \{e_1, ..., e_n\}$
Ensure: $\Phi_{\mathfrak{E}}$ is the set of all the maximal component patterns of $\mathfrak{E}$
$\wedge \mathscr{E}_{\mathfrak{E}}$ is the set of the structurally similar subsets of $\mathfrak{E}$

```
1: procedure BUILDPATTERNSSET(𝔈, Φ_𝔈, ℰ_𝔈)
2:     Φ_𝔈 ← ∅
3:     ℰ_𝔈 ← ∅
4:     for i = 1 to n do ▷ is there already a subset containing the current entity type?
5:         if ∃ ℰ_k ∈ ℰ_𝔈 : e_i ∈ ℰ_k then
6:             ℰ_i ← ℰ_k
7:         else
8:             ℰ_i ← {e_i}
9:             ℰ_𝔈 ← ℰ_𝔈 ∪ {ℰ_i}
10:        end if
11:        for j = i + 1 to n do ▷ for all the remaining entity types...
12:            if ∃ ℰ_l ∈ ℰ : e_j ∈ ℰ_l then
13:                ℰ_j ← ℰ_l
14:            else
15:                ℰ_j ← {e_j}
16:            end if
17:            ℰ_𝔈 ← ℰ_𝔈 \ ℰ_i
18:            ℰ_𝔈 ← ℰ_𝔈 \ ℰ_j
19:            if e_i ⇜⇝^n e_j ∧ n > 0 then ▷ e_i and e_j are structurally similar
20:                ℰ_i ← ℰ_i ∪ ℰ_j
21:                ℰ_𝔈 ← ℰ_𝔈 ∪ {ℰ_i}
22:            else
23:                ℰ_𝔈 ← ℰ_𝔈 ∪ {ℰ_i} ∪ {ℰ_j}
24:            end if
25:            Φ_𝔈 ← Φ_𝔈 ∪ {ρ̃_{i,j}} ▷ store the maximal pattern anyway
26:        end for
27:    end for
28: end procedure
```

7.2.3 Submitting structural ambiguities to the end-users for arbitration

Once the sets $\Phi_{\mathfrak{E}} = \{\tilde{\rho}_{i,j}\}$ and $\mathscr{E}_{\mathfrak{E}}$ have been built from all the entity types contained in $\mathcal{S}$, we can visually point out the discovered similarities in the user-drawn interfaces, in order ask the end-users to to validate or reject them, as in Fig. 7.6, Fig. 7.7 and Fig. 7.8.

The validation process therefore consists in examining each maximal pattern $\tilde{\rho}_{i_j,i_k}$ discovered and its associated pair of entity types e_{i_j} and e_{i_k} with regards to the structurally similar subset $\mathscr{E}_i \subseteq \mathscr{E}_{\mathfrak{E}}$ to which they belong, in order to

specify the nature of their relation, as defined in Section 7.2.1:

- e_{i_j} equals e_{i_k} (noted $e_{i_j} \overset{e}{\rightleftarrows} e_{i_k}$)
- e_{i_j} specialises e_{i_k} (noted $e_{i_j} \overset{s}{\rightrightarrows} e_{i_k}$, or conversely $e_{i_k} \overset{s}{\rightrightarrows} e_{i_j}$)
- e_{i_j} unites with e_{i_k} (noted $e_{i_j} \overset{u}{\rightleftarrows} e_{i_k}$)
- e_{i_j} complements e_{i_k} (noted $e_{i_j} \overset{c}{\rightrightarrows} e_{i_k}$, or conversely $e_{i_k} \overset{c}{\rightrightarrows} e_{i_j}$)
- difference : $e_{i_j} \overset{e}{\not\rightleftarrows} e_{i_k}$

Defining semantically equivalent subsets of entity types

For this purpose, we first need to ask the end-users to elicit the *semantically equivalent* subsets of $\mathscr{E}_i$ and assigning them a unifying term. By doing so, we actually define the subsets of entity types that are *equal*, as well as the concepts they represent. More formally, we want the end-users to specify $\mathscr{E}_i'$ so that it concurs with the following definition:

Definition 7.20. Given a set $\mathscr{E}_i$ of structurally equivalent entity types, the set $\mathscr{E}_i^{eq} = \{\mathscr{E}_{i_1}^{eq}, \mathscr{E}_{i_2}^{eq}, ...\}$ is the set of the *semantically equivalent* subsets of $\mathscr{E}_i$ iff;

$$\begin{aligned}
&\forall i,j,k \quad : \mathscr{E}_{i_j}^{eq} \cap \mathscr{E}_{i_k}^{eq} &&= \emptyset \\
&\forall i \qquad\;\; : \bigcup_j \mathscr{E}_{i_j}^{eq} &&= \mathscr{E}_i \\
&\forall i,j,k,l : e_{i_{j_k}}, e_{i_{j_l}} \in \mathscr{E}_{i_j}^{eq} \Leftrightarrow e_{i_{j_k}} \overset{e}{\rightleftarrows} e_{i_{j_l}} \quad \text{(i.e. } e_{i_{j_k}} \equiv e_{i_{j_l}}\text{)}
\end{aligned}$$

⌟

Note that these $\mathscr{E}_i^{eq}$ and their associated term $t_{i_j}^{eq}$ defines a partition of $\mathfrak{E}$ into $\Upsilon^{eq} = \{(t_{i_j}^{eq}, \mathscr{E}_{i_j}^{eq})\}$, so that :

$$\begin{aligned}
&\forall i,j,k \;:\; t_{i_j}^{eq} \neq t_{i_k}^{eq} \\
&\forall i,j,k \;:\; \mathscr{E}_{i_j}^{eq} \cap \mathscr{E}_{i_k}^{eq} = \emptyset \\
&\bigcup_{i,j} \mathscr{E}_{i_j}^{eq} = \mathfrak{E}
\end{aligned} \tag{7.7}$$

If we consider again the set $\mathfrak{E}$ containing the entity types highlighted in Fig. 7.9, we could for instance obtain the following subsets:

- $\mathscr{E}_1^{eq} = \{$ {`Customer`}, {`Order`} $\}$
- $\mathscr{E}_2^{eq} = \{$ {`Address`, `Address (alternative)`, `Location`}, {`Provider`} $\}$
- $\mathscr{E}_3^{eq} = \{$ {`Product`}, {`Special good`}, {`Service`} $\}$

with $t_{2_1}^{eq}$ = “Address”.

Defining union of subsets of entity types

Once these concepts are defined, we proceed with the specification of the possible *unions* between pairs of concepts conveyed by any $\mathscr{E}_{i_j}^{'}, \mathscr{E}_{i_k}^{'} \in \mathscr{E}_i^{'}$, and we assign a term to their underlying parent concept. More formally, we want the end-users to specify $\mathscr{E}_i^{un}$ so that it concurs with the following definition:

Definition 7.21. Given a set $\mathscr{E}_i$ of structurally equivalent entity types and its associated set $\mathscr{E}_i^{eq}$ of semantically equivalent subsets, the set $\mathscr{E}_i^{un} = \{\mathscr{E}_{i_1}^{un}, \mathscr{E}_{i_2}^{un}, ...\}$ is the set of the unions of semantically equivalent subsets of $\mathscr{E}_i$ iff;

$$\begin{aligned}
&\forall i,j,k \quad : \mathscr{E}_{i_j}^{un} \cap \mathscr{E}_{i_k}^{un} \quad = \emptyset \\
&\forall i \quad\quad\ : \bigcup_j \mathscr{E}_{i_j}^{un} \quad\quad = \mathscr{E}_i^{eq} \\
&\forall i,j,k,l : e_{i_{j_k}}, e_{i_{j_l}} \in \mathscr{E}_{i_j}^{un} \Leftrightarrow e_{i_{j_k}} \overset{u}{\rightleftarrows} e_{i_{j_l}}
\end{aligned}$$

⌟

Note that these $\mathscr{E}_i^{un}$ and their associated term $t_{i_j}^{un}$ defines a partition of $\mathfrak{E}$ into $\Upsilon^{un} = \{(t_{i_j}^{un}, \mathscr{E}_{i_j}^{un})\}$, so that :

$$\begin{aligned}
&\forall i,j,k \ : \ t_{i_j}^{un} \ \neq \ t_{i_k}^{un} \\
&\forall i,j,k \ : \ \mathscr{E}_{i_j}^{un} \cap \mathscr{E}_{i_k}^{un} = \emptyset \\
&\bigcup_{i,j} \mathscr{E}_{i_j}^{un} = \mathfrak{E}
\end{aligned} \tag{7.8}$$

For the set $\mathfrak{E}$ of the entity types highlighted in Fig. 7.9, we could for instance obtain the following subsets:

- $\mathscr{E}_1^{un} = \{\{\{\texttt{Customer}\}\}, \{\{\texttt{Order}\}\}\}$
- $\mathscr{E}_2^{un} = \{\{\{\texttt{Address}, \texttt{Address(alternative)}, \texttt{Location}\}\}, \{\{\texttt{Provider}\}\}\}$
- $\mathscr{E}_3^{un} = \{\{\{\texttt{Product}\}, \{\texttt{Service}\}\}, \{\{\texttt{Special good}\}\}\}$

with $t_{3_1}^{eq}$ = "Solution".

Defining specialisation among subsets or unions of entity types

Then, we carry on by eliciting the possible *specialisations* between pairs of concepts conveyed by any $\mathscr{E}_{i_j}^{'}, \mathscr{E}_{i_k}^{'} \in \mathscr{E}_i^{'}$ or their parent concept. More formally, we want the end-users to specify $\mathscr{E}_i^{sp}$ so that it concurs with the following definition:

Definition 7.22. Given a set $\mathscr{E}_i$ of structurally equivalent entity types, its associated sets $\mathscr{E}_i^{eq}$ and $\mathscr{E}_{i_1}^{un}$, the set $\mathscr{E}_i^{sp} = \{(\alpha_{i_1}, \beta_{i_1}), (\alpha_{i_2}, \beta_{i_2}), ...\}$ is the set of the specialisations of $\mathscr{E}_i$ iff;

$$\begin{aligned} &\forall i, j \quad : (\alpha_{i_j} \in \mathscr{E}_i^{eq} \vee \alpha_{i_j} \in \mathscr{E}_i^{un}) \wedge (\beta_{i_j} \in \mathscr{E}_i^{eq} \vee \beta_{i_j} \in \mathscr{E}_i^{un}) \\ &\forall i, j, k, l : e_{i_{j_k}} \in \alpha_{i_j},\ e_{i_{j_l}} \in \beta_{i_j} \quad \Leftrightarrow e_{i_{j_k}} \overset{s}{\rightrightarrows} e_{i_{j_l}} \end{aligned}$$

⌟

For the set $\mathfrak{E}$ of the entity types highlighted in Fig. 7.9, we could for instance obtain the following subsets:

- $\mathscr{E}_1^{sp} = \emptyset$
- $\mathscr{E}_2^{sp} = \emptyset$
- $\mathscr{E}_3^{sp} = \{(\{\texttt{Special good}\}, \{\texttt{Product}\})\}$

Defining complementarity among subsets or unions of entity types

And finally, we end by specifying the possible *complementarities* between pairs of concepts conveyed by any $\mathscr{E}_{i_j}^{'}$, $\mathscr{E}_{i_k}^{'} \in \mathscr{E}_i^{'}$ or their parent concept. More formally, we want the end-users to specify $\mathscr{E}_i^{cp}$ so that it concurs with the following definition:

Definition 7.23. Given a set $\mathscr{E}_i$ of structurally equivalent entity types, its associated sets $\mathscr{E}_i^{eq}$ and $\mathscr{E}_{i_1}^{un}$, the set $\mathscr{E}_i^{cp} = \{(\alpha_{i_1}, \beta_{i_1}), (\alpha_{i_2}, \beta_{i_2}), ...\}$ is the set of the complementarities of $\mathscr{E}_i$ iff;

$$\begin{aligned} &\forall i, j \quad : (\alpha_{i_j} \in \mathscr{E}_i^{eq} \vee \alpha_{i_j} \in \mathscr{E}_i^{un}) \wedge (\beta_{i_j} \in \mathscr{E}_i^{eq} \vee \beta_{i_j} \in \mathscr{E}_i^{un}) \\ &\forall i, j, k, l : e_{i_{j_k}} \in \alpha_{i_j},\ e_{i_{j_l}} \in \beta_{i_j} \quad \Leftrightarrow e_{i_{j_k}} \overset{c}{\rightrightarrows} e_{i_{j_l}} \end{aligned}$$

⌟

For the set $\mathfrak{E}$ of the entity types highlighted in Fig. 7.9, we could for instance obtain the following subsets:

- $\mathscr{E}_1^{cp} = \{(\{\texttt{Order}\}, \{\texttt{Customer}\})\}$
- $\mathscr{E}_2^{cp} = \{(\{\texttt{Provider}\}, \{\texttt{Address}, \texttt{Address(alternative)}, \texttt{Location}\})\}$
- $\mathscr{E}_3^{cp} = \emptyset$

Preventing stalemates

One of the major risks during this process is to gather conflictual or problematic decisions that would lead to a stalemate. Consider for instance three

entity types e_1, e_2 and e_3. Declaring that e_1 and e_2 are equivalent, but that e_1 specialises e_3 while e_2 is specialised by e_3 intuitively creates a puzzling situation with a hierarchical cycle. Fig. 7.12 illustrates such a situation, with `Individual` and `Person` being equivalent, while respectively specialising and being specialised by `Customer`.

Figure 7.12: A problematic situation where two entity types are equivalent (`Individual` and `Person`), but respectively specialises and is specialised by a third one (`Customer`).

Detection mechanisms can obviously be set to detect these kinds of situation. However, this also highlights once again the primordial role of the analyst in our approach, as he is the most suited person to notice and prevent such cases. He should therefore help the end-users to avoid them by guiding him into structuring their decisions in the most consistent fashion.

7.2.4 Processing the structural decisions of the end-users

Similarly to the semantic analysis, the final step of this validation consists in updating the widgets, the schemas and the mappings of thesaurus τ_S for each type of validated similarity. The main aspect concerns the pre-integration of each individual schema into a single schema based on these validated redundancies.

First of all, we process the sets $\mathscr{E}_{i_j}^{eq}$ of *equal* entity types. Whenever these sets contain more that one element, a supertype is created and assigned the unifying term. If a set contains only one entity type, this entity type is considered the supertype of $\mathscr{E}_{i_1}^{eq}$ and is also assigned the unifying term if it defers from its original term.

Then, we process the sets $\mathscr{E}_{i_1}^{un}$ of *united* entity types. Whenever these sets contain more that one element, a supertype is also created and assigned the unifying term. If a set contains only one entity type, this entity type is considered the supertype of $\mathscr{E}_{i_1}^{un}$ and is also assigned the unifying term if it differs from its original term.

Afterwards, the $(\alpha_{i_j}, \beta_{i_j}) \in \mathscr{E}_i^{sp}$ are processed. An IS-A relationship is created between the supertype of α_{i_j} and the supertype of β_{i_j}.

Subsequently, the $(\alpha_{i_j}, \beta_{i_j}) \in \mathscr{E}_i^{cp}$ are processed. A stereotyped relationship type is created between the supertype of α_{i_j} and the supertype of β_{i_j}. Alternatively, this step can be replaced by looping back and modifying the container holding the referential elements, so that the latter are moved into a new sub container associated with a given term of the referred element.

Finally, thesaurus τ_S is updated. The label of each container associated to an entity type involved in a validated structural similarity may also be updated if relevant, such as the label "Location" that may be updated to Address.

The whole validation process can hence be described by Algorithm 7.8, and as can be observed, it implies that all the *semantically equivalent* entity types of the schemas now have a super type, and that the entity types are now hierarchically structured. If cycles should appear in the schemas in spite of the analysts attention, they redundancies between the involved entity types should be re-examined to prevent these cycles.

Fig. 7.13 illustrates the update of the forms, while Fig. 7.14 illustrates the update of the schemas for our running example. As one can notice, at this point of the process, the appearance of the forms hasn't changed much, unlike their underlying schemas that have been pre-integrated. As we can also see, the components still need to be properly integrated and "transferred" to the appropriate supertypes.

7.2.5 Reducing structural redundancies

As for semantic analysis, structural redundancies can be reduced upstream. This can be managed in the drawing phase by providing common predefined and standardised *reusable patterns* (typically, such as an `Address` or a `Person`), having a direct RSFM representation and an associated GER counterpart. Such reusable constructs could for instance be inferred from existing ontologies or patterns classifications (such as Coad's *object-oriented patterns* [Coad, 1992]), or defined from our own elicited *patterns*.

7.3 Output

At the end of this interactive process, we obtain a pre-integrated schema s resulting from the terminological and structural analysis of the set of schemas $\mathcal{S}$ obtained through the *Adapt* phase. In this schema, the terminology has been unified so that every element associated with a given term now represent the same concept. Also, the sub schemas originally associated with each form are now connected through the relationship types and IS-A hierarchies of their

Algorithm 7.8 ValidateStructuralSimilarities : Validate the semantically similar subsets of a given thesaurus (1/2)

Require: $\mathcal{S}$
Ensure: s is the pre-integrated schema of all the schemas of $\mathcal{S}$, and τ is his thesaurus

```
1: procedure ValidateStructuralSimilarities(S, s, τ)
2:     𝔈 ← {e_i | (∃ s_j ∈ S : e_i ∈ s_j) ∧ (getType(e_i) = ENTITYTYPE)}
3:     buildPatternsSet(𝔈, Φ_𝔈, 𝒪_𝔈)
4:     s ← createSchema()
5:     for all s_i ∈ S do
6:         copy s_i in s
7:     end for
8:     𝒯 ← ∅
9:     for all 𝒪_i ∈ 𝒪_𝔈 do
10:        Ask> define: 𝒪_i^eq
11:        for all 𝒪_ij^eq ∈ 𝒪_i^eq do
12:            t_ij^eq ← null
13:            while t_ij^eq = null ∨ t_ij^eq ∈ 𝒯 do
14:                Ask> define: t_ij^eq  ▷ define: a unifying term for the supertype
15:            end while
16:            𝒯 ← 𝒯 ∪ {t_ij^eq}
17:        end for
18:        Ask> define: 𝒪_i^un
19:        for all 𝒪_ij^un ∈ 𝒪_i^un do
20:            t_ij^un ← null
21:            while t_ij^un = null ∨ t_ij^un ∈ 𝒯 do
22:                Ask> define: t_ij^un  ▷ define: a unifying term for the supertype
23:            end while
24:            𝒯 ← 𝒯 ∪ {t_ij^un}
25:        end for
26:        Ask> define: 𝒪_i^sp
27:        Ask> define: 𝒪_i^cp
28:        for all 𝒪_ij^eq ∈ 𝒪_i^eq do
29:            entityType_i ← createEntityType(s, t_ij^eq)
30:            for all e_{i_{j_k}} ∈ 𝒪_ij^eq do
31:                createIsA(entityType_i, e_{i_{j_k}})  ▷ entityType_i is a supertype for e_{i_{j_k}}
32:            end for
33:        end for
34:        for all 𝒪_ij^eq ∈ 𝒪_i^un do
35:            entityType_i ← createEntityType(s, t_ij^un)
36:            for all e_{i_{j_k}} ∈ 𝒪_ij^un do
37:                createIsA(entityType_i, e_{i_{j_k}})  ▷ entityType_i is a supertype for e_{i_{j_k}}
38:            end for
39:        end for
```

▷ [continued on page 125]

Algorithm 7.9 ValidateStructuralSimilarities (2/2)

for all $(\alpha_{i_j}, \beta_{i_j}) \in \mathscr{E}_i^{sp}$ **do**
  $e_{i_{j_1}} \leftarrow superType(\alpha_{i_j})$
  $e_{i_{j_2}} \leftarrow superType(\beta i_j)$
  $createIsA(e_{i_{j_2}}, e_{i_{j_1}})$ ▷ $e_{i_{j_2}}$ is a supertype for $e_{i_{j_1}}$
end for
for all $(\alpha_{i_j}, \beta_{i_j}) \in \mathscr{E}_i^{cp}$ **do**
  $e_{i_{j_1}} \leftarrow superType(\alpha_{i_j})$
  $e_{i_{j_2}} \leftarrow superType(\beta i_j)$
  $relType_i \leftarrow createRelationshipType(e_{i_{j_1}}, e_{i_{j_2}})$
  setCardinalities$(relType_i, 1, 1, 0, N)$
  setStereotype$(relType_i, \texttt{refersTo})$
end for
end for
BUILDTHESAURUS$(\{s\}, \tau)$
end procedure

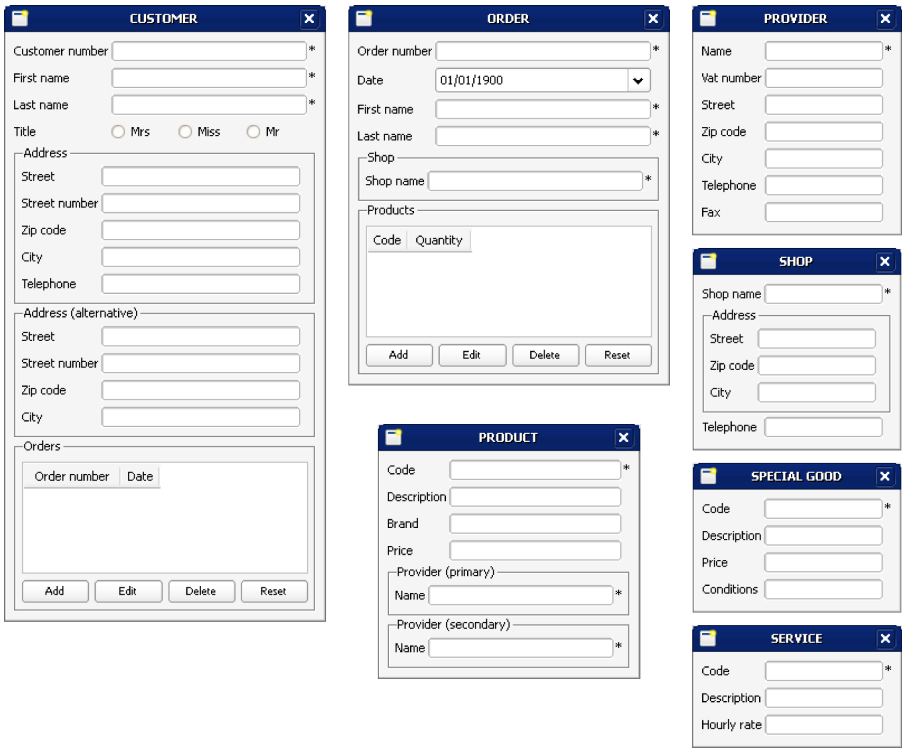


Figure 7.13: The updated forms of the running example after validation of structural redundancies.

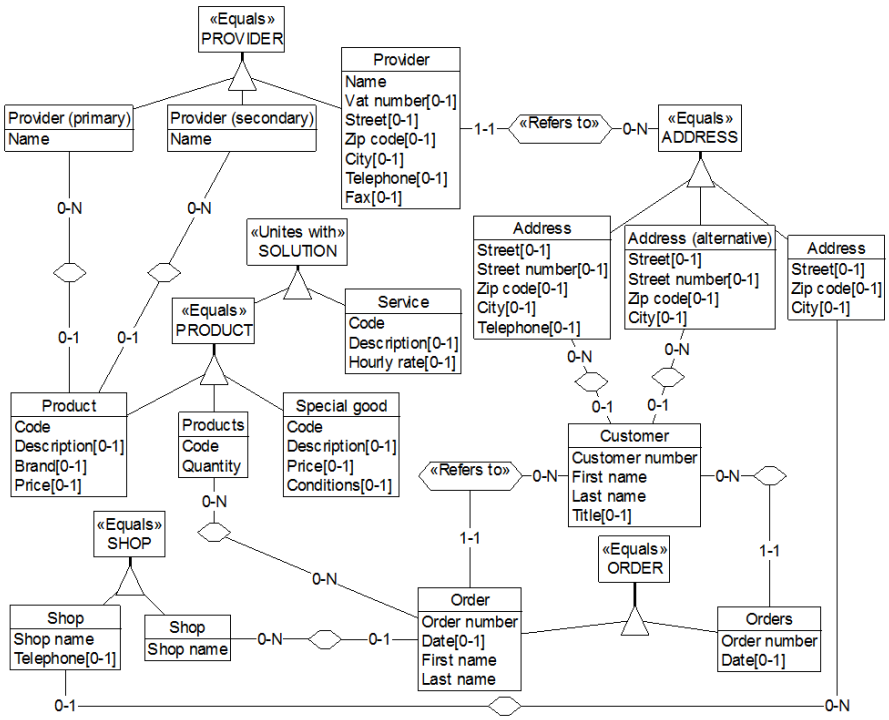


Figure 7.14: The pre-integrated schema of the example after validation of structural redundancies. The newly created supertypes and relationship types are marked with a stereotype expressing their meaning: "`Equals`" stands for equality, "`Unites with`" for union, "`Refers to`" for complementarity. There is no stereotype for the specialisation, as it is implicit.

entity types. Besides, the maximal component pattern between each pair of entity type is stored for further processing.

The newly created "parent" entity types still need to be complemented by the appropriate attributes shared by their "children" entity types, and the stereotyped relationship types must also be supplemented with the relevant referential attributes. This process, whose responsibility rests with the *Bind* step, could typically lead to merging the components of the entity types involved in the same IS-A relation.

Chapter 8

NURTURE

Eliciting dependencies and constraints

The previous chapter dealt with the analysis of terminological and structural ambiguities within a set of schemas in order to pre-integrate the latter into a single schema with a unified terminology. In order to enrich this schema, we now need to discover additional constraints and dependencies on its elements.

Though these constraints can be provided directly, it appears that the acquisition and use of data samples may also be useful and more natural in this process. Indeed, not only do data samples test the ability of the user-drawn form-based interfaces to gather the necessary information, but it also helps to visualise the implications of existing constraints. Moreover, their analysis may in turn reveal possible unsuspected constraints.

In this chapter, we therefore formalise the notions of data samples, constraints and dependencies, then present an interactive process inspired by the principles of Armstrong relations, in order to acquire data samples that will restrict the possible "hidden" constraints, and to arbitrate constraints that will in turn restrict the tuples that can be encoded.

8.1 Delimiting constraints and dependencies

There are numerous types of constraints and dependencies that can be established for a given schema. They can concern individual elements, their components, or even how (the components of) an element can affect (the components of) other elements. In this doctoral research, we focus on constraints

that can be expressed for entity types and their components, developing an approach that could be intuitively extended to constraints and dependencies among (components of) multiple elements.

More specifically, for each entity type of the pre-integrated schema s, we want to elicit the constraints and dependencies presented in Section 3.4.2, which we group as follows:

- *technical constraints*, which define the following restrictions on the individual components of each entity type:
  - the *minimal and maximal cardinalities*;
  - the *value type*;
  - the *value size*;
  - the *prerequisite optional components*, if the component is optional;
- *existence constraints*, which define how the optional components should coincide for each entity type;
- *functional dependencies*, which define the implications between sets of components (we do not treat multivalued dependencies);
- *identifiers*, which define the sets of components that uniquely identify a given instance of a given entity type.

Some of these properties can be trivial and may be expressed directly, or have been expressed during the drawing step or subsequent modifications of the original form-based interfaces. For instance, in the forms of Fig. 7.13, the `Title` of a `Customer` appears to be optional and single-valued, and the `Zip Code` of a `Provider` may have been encoded as a textual value.

However, the specified properties may need to be refined, and there may be some unsuspected constraints and dependencies among the elements of the schema. As we have seen, we can take advantage of data samples to induce possible implicit constraints and mine functional dependencies. For instance, we could observe that though optional, the `Title` is systematically filled for each encoded `Customer`, or that the `Zip Code` of a `Provider` is always encoded using only numerical characters. We could also observe, for example, that there is always at least a `Zip Code` or a `City` for each encoded `Provider`. These observations must be submitted to end-users form arbitration, in order to eventually enrich the pre-integrated schema.

Unfortunately, we observed that the existing approaches rely on massive pre-existing data sets, which is here problematic. Indeed given our context, there is possibly no available data samples, or their re-encoding would be too expensive. It is anyway unrealistic to ask end-users to willingly provide numerous data samples. This naturally calls for new ways to discover and suggest

constraints and dependencies on-the-fly, based on the incremental input of data samples by the end-users.

Before introducing our approach to suggest constraints and dependencies, let us start by formalising the notions of data samples and tuples, as well as these constraints and dependencies.

8.2 Formalising data samples and tuples

First of all, the easiest way to ask end-users to provide data samples is to let them use the very form-based interfaces they drew as an encoding means, knowing that each form is associated with several entity types. Now recall that we presented the relational model of a database in Section 3.4.2, and let us adapt it for our purpose.

A *relation* (or *table*) is the natural equivalent of an entity type in the relational realm, while its *attributes* (or *columns*) can be associated to the components of the entity type. Since a *tuple* (or *row*) contains a value for each attribute of the relation with respect to its domain, we can likewise define a tuple for an entity type.

Let $\mathcal{A}$ and $\mathcal{R}$ respectively be the set of simple attributes and roles of a given entity type e, and let $\mathcal{T}_e$ be the set of tuples associated with it. Recall also that e_{C_i} represents the entity type associated to e through the relationship involving the role C_i (see Section 7.2.1).

Definition 8.1. Given an entity type e and its set of components $\mathcal{C} = \mathcal{A} \cup \mathcal{R}$, t is a *tuple* for e iff:

$$t = \{(C_i, v_i) \mid (\bigcup_i C_i = \mathcal{C}) \wedge ((C_i \in \mathcal{A} \wedge v_i \in Dom(C_i)) \vee (C_i \in \mathcal{R} \wedge vi \in \mathcal{T}_{e_{C_i}}))\}$$

⌟

Furthermore:

Definition 8.2. The tuple t' is the projection of the tuple t on the set of components $\mathcal{C}$ (noted $t[\mathcal{C}]$) iff: $t' = \{(C_i, v_i) \in t \mid C_i \in \mathcal{C}\}$ ⌟

The projection of a tuple t on a single component C (noted $t[C]$) is therefore the pair (C_i, v_i) verifying $C_i = C$, and it can be null (which is noted $\varnothing$ and implies that the component C is optional and that no value has been provided for it) or multivalued.

Whenever an end-user provides a data sample, he actually transparently provides a tuple for each entity type associated with the form-based interface. In this doctoral research, we focus and reason on *valid* user-provided data

samples, i.e. data samples that are realistic and consistent with the current state of the requirements acquired using the RAINBOW approach. However, it would also be possible to reason on *invalid* data samples, which would also imply detailing the criteria for invalidity.

Recalling that $\mathfrak{E}_F$ is the set of entity types corresponding to a given form (see Section 6.3), we can hence formally define a data sample as follows:

Definition 8.3. A set $d = \{(e_i, t_i)\}$ is a *data sample* for the form F iff:

$$(\bigcup_i e_i = \mathfrak{E}_F) \wedge (\forall i : t_i \text{ is a tuple for } e_i)$$

⌟

Consider for instance the form `Product` illustrated in Fig. 7.13. A data sample for this form would actually provide tuples for the entity types `Product`, `Provider (primary)` and `Provider (secondary)` of Fig. 7.14.

Let us now formalise the notions of constraints and dependencies for an entity type and its associated set of tuples.

8.3 Formalising constraints and dependencies

8.3.1 Technical constraints

Let us consider a given entity type e, having the set of components $\mathcal{C} = \mathcal{A} \cup \mathcal{R}$, the set of optional components $\mathcal{C}' \subseteq \mathcal{C}$ and the set of tuples $\mathcal{T}_e$. For a given component $C \in \mathcal{C}$, the different types of technical constraints that we inquire about can be grouped in the set $\mathbb{T} = \{$`cardinality`, `value type`, `value size`, `prerequisite components`$\}$.

The domains of values for each type of these technical constraints can be derived from the specification of the Simplified Form Model as follows:

- $Dom($`cardinality`$) = \{(0,1), (1,1), (0,N), (1,N)\}$
- $Dom($`value type`$) = \{$`text`, `integer`, `real`, `boolean`, `date`$\}$ if $C \in \mathcal{A}$
- $Dom($`value type`$) = \emptyset$ if $C \in \mathcal{R}$
- $Dom($`value size`$) = \mathbb{N}$ if $C \in \mathcal{A}$
- $Dom($`value size`$) = \emptyset$ if $C \in \mathcal{R}$
- $Dom($`prerequisite`$) = \{\mathcal{C}'' \subset \mathcal{C}' \mid C \notin \mathcal{C}'\}$

We can therefore formally define a technical constraint as follows:

Definition 8.4. Given an entity type e and its set of components $\mathcal{C}$, $\theta_i = (C_i, p_i, v_i)$ is a *technical constraint* for e iff : $C_i \in \mathcal{C} \wedge p_i \in \mathbb{T} \wedge v_i \in Dom(p_i)$ ⌟

Furthermore:

Definition 8.5. A tuple $t \in \mathcal{T}_e$ *agrees* with a technical constraint $\theta = (C, p, v)$ (noted $t \bowtie \theta$), iff the projection $t[C]$ respects the value v for the property p. ⌟

Generally speaking, a tuple agrees with a technical constraint if the latter isn't too restrictive regarding the value(s) of the tuple for the given component. Typically, the tuple must provide at least a value if the constraints specifies a mandatory cardinality, and at most a value if the constraint specifies a single-valued cardinality.

Similarly, if the component is an attribute, the value(s) associated with the component can either be of any type if the constraint specifies a textual value type, or integer if the constraint specifies a real value type. In any other cases, the value type of the value(s) associated with the component must absolutely concur with the constraint.

Also, the size of the value(s) associated with the component must be smaller or equal that the value size constraint, given that the component is an attribute.

And finally, if the value(s) associated with the component is not null, the values associated with the prerequisite components cannot be null. More formally, a tuple t therefore agrees with a technical constraints $\theta = (C, p, v)$ when:

- C is a component, $p =$ `cardinality`, $v = (mincard, maxcard)$ and $mincard \leq t[C] \leq maxcard$
- C is an attribute, $p =$ `value type`, and :
  - $v = text$ and the value type of $t[C]$ is in $Dom($`value type`$)$
  - $v = real$ and the value type of $t[C]$ is in $\{$`integer`, `real`$\}$
  - $v \in Dom($`value type`$)\backslash\{text, real\}$ and the value type of $t[C]$ is equal to v
- C is a role, $p =$ `value type`, and $v = \varnothing$
- C is an attribute, $p =$ `value size`, and the size of $t[C]$ is $\leq v$
- C is a role, $p =$ `value size`, and $v = \varnothing$
- C is a component, $p =$ `prerequisite` and $t[C] \neq \varnothing \Rightarrow \forall C_i \in v : t[C_i] \neq \varnothing$

Consequently:

Definition 8.6. A technical constraint θ is *satisfied* by a set of tuples $\mathcal{T}$ (noted $\mathcal{T} \models \theta$) iff: $\forall t_i \in \mathcal{T} : t_i \bowtie \theta$. ⌟

We call Θ_e the set of all the technical constraints defined on a given entity type e and satisfied on its set of tuples $\mathcal{T}_e$, so that:

$$\forall C_i \in \mathcal{C}, p_j \in \mathbb{T} \ : \ \exists!\ \theta = (C_i, p_j, v_k) \in \Theta_e \tag{8.1}$$

8.3.2 Existence constraints

Let us consider a given entity type e, having the set of optional components $\mathcal{C}'$ and the set of tuples $\mathcal{T}_e$. For a given set of optional components $\mathcal{X} \subseteq \mathcal{C}'$, the different types of existence constraints that it may support can be grouped in the set $\mathbb{E} = \{\texttt{coexistence}, \texttt{exactly one}, \texttt{at most one}, \texttt{at least one}\}$.

We can therefore formally define an existence constraint as follows:

Definition 8.7. Given an entity type e and its set of optional components $\mathcal{C}'$, $\xi_i = (\mathcal{X}_i, p_i)$ is a *existence constraint* for e iff: $\mathcal{X}_i \subseteq \mathcal{C}' \wedge p_i \in \mathbb{E}$. ⌟

Furthermore:

Definition 8.8. A tuple $t \in \mathcal{T}_e$ *agrees* with a existence constraint $\xi = (\mathcal{X}, p)$ (noted $t \bowtie \xi$), iff the projection $t[\mathcal{X}]$ respects the constraint p. ⌟

Practically, a tuple t will agree with an existence constraint $\xi = (\mathcal{X}, p)$ if:

- $p = \texttt{coexistence}$ and $(\forall\ C_i \in \mathcal{X} : t[C_i] = \varnothing) \vee (\forall\ C_i \in \mathcal{X} : t[C_i] \neq \varnothing)$
- $p = \texttt{exactly one}$ and $\exists!\ C_i \in \mathcal{X} : t[C_i] \neq \varnothing$
- $p = \texttt{at least one}$ and $\exists\ C_i \in \mathcal{X} : t[C_i] \neq \varnothing$
- $p = \texttt{at most one}$ and $(\forall\ C_i \in \mathcal{X} : t[C_i] = \varnothing) \vee (\exists! C_i \in \mathcal{X} : t[C_i] \neq \varnothing)$

Consequently:

Definition 8.9. A existence constraint ξ is *satisfied* by a set of tuples $\mathcal{T}$ (noted $\mathcal{T} \models \xi$) iff: $\forall t_i \in \mathcal{T} : t \bowtie \xi$. ⌟

We call Ξ_e the set of all the existence constraints defined on a given entity type e and satisfied on its set of tuples $\mathcal{T}_e$.

8.3.3 Functional dependencies

Let us consider a given entity type e, having the set of components $\mathcal{C}$ and the set of tuples $\mathcal{T}_e$. We can formally define a functional dependency as follows:

Definition 8.10. Given an entity type e and its set of components $\mathcal{C}$, a *functional dependency* over e is an expression $f : \mathcal{L} \to \mathcal{R}$, with $\mathcal{L}, \mathcal{R} \subseteq \mathcal{C}$, restricting the possible tuples of e. ⌟

We respectively call $\mathcal{L}$ and $\mathcal{R}$ the *left-hand side* and *right-hand side* of the functional dependency f. Furthermore:

Definition 8.11. A functional dependency $f : \mathcal{L} \to \mathcal{R}$, is *satisfied* for a set of tuples $\mathcal{T}$ (noted $\mathcal{T} \models f$), iff: $\forall t_i, t_j \in \mathcal{T}\ :\ t_i[\mathcal{L}] = t_j[\mathcal{L}] \Rightarrow t_i[\mathcal{R}] = t_j[\mathcal{R}]$ ⌟

We call $\mathfrak{F}_e$ the set of all the functional dependencies defined on a given entity type e and satisfied on its set of tuples $\mathcal{T}_e$. Besides, recall that functional dependencies can be compared using Armstrong's axioms, which were introduced in Section 3.4.2.

8.3.4 Unique constraints

Let us consider a given entity type e, having the set of components $\mathcal{C}$, the set of mandatory components $\mathcal{C}^* \subseteq \mathcal{C}$ and the set of tuples $\mathcal{T}_e$. We can formally define a unique constraint as follows:

Definition 8.12. Given an entity type e, its set of components $\mathcal{C}$, its set of mandatory components $\mathcal{C}^* \subseteq \mathcal{C}$ and its set of tuples $\mathcal{T}_e$, the set of components $\mathcal{X} \subseteq \mathcal{C}^*$ is an *identifier* for e iff: $\forall t_i \in \mathcal{T}_e,\ \nexists t_j \neq t_i \in \mathcal{T}_e\ :\ t_j[\mathcal{X}] = t_i[\mathcal{X}]$ ⌟

Besides, the different types of unique constraints can be grouped in the set $\mathbb{U} = \{\texttt{primary}, \texttt{secondary}\}$, so that:

Definition 8.13. Given an entity type e, its set of components $\mathcal{C}$, its set of mandatory components $\mathcal{C}^* \subseteq \mathcal{C}$ and its set of tuples $\mathcal{T}_e$, $\upsilon_i = (\mathcal{X}_i, p_i)$ is a *unique constraint* for e iff: $\mathcal{X}_i \subseteq \mathcal{C}^* \wedge p_i \in \mathbb{U}$. ⌟

Furthermore:

Definition 8.14. A unique constraint $\upsilon = (\mathcal{X}, p)$ is *satisfied* by a set of tuples $\mathcal{T}_e$ (noted $\mathcal{T}_e \models \upsilon$) iff $\mathcal{X}$ is an identifier for the entity type e. ⌟

We call Υ_e the set of all the unique constraints defined on a given entity type e and satisfied on its set of tuples $\mathcal{T}_e$.

8.4 Managing the process

8.4.1 Overview

Now that we have formalised the notions of data samples, constraints and dependencies, let us expose our nurturing process, considering that we initially have no available tuples at all. Recall that our objective is to involve then end-users in the elicitation of constraints and dependencies, while ensuring that none of them are forgotten. We therefore propose to start by envisaging initial possible constraints and dependencies. Then, using user input, we progressively *enforce* (i.e. validate) or *discarded* (i.e. refute) them, and generate alternatives until they are all arbitrated. This process hence relies on several sub processes:

- the initialisation of all initial valid constraints and dependencies;

- the acquisition and analysis of new valid data samples in order to automatically discard the invalid constraints and dependencies, and possibly generate alternatives;
- the arbitration of currently valid constraints and dependencies through user input, and the subsequent generation of alternatives;
- the processing of the enforced constraints and dependencies, once there are no other valid constraints or dependencies left.

Note that the acquisition of data samples will progressively restrict the set of possibly valid constraints, and that conversely, enforcing constraints will also restrict the future data samples that will be encodable. Let us now examine each of the sub processes.

8.4.2 Initialisation

Before beginning the interaction with the end-users, we start by initialising an empty set of tuples and defining the initial sets of enforced, valid and discarded constraints and dependencies for each entity type e. The valid sets will be used to provide suggestions to the users, while the discarded sets will be used to collect all the rejected constraints and dependencies.

Note that in the algorithms of this section, we use simplified methods of the form `get<PropertyName>`(c) and `set<PropertyName>`$(c, value)$ to access the different components properties for the sake of simplicity. These methods transparently access the necessary (meta) properties for these components.

Technical constraints

Let Θ_e, $\check{\Theta}_e$ and $\bar{\Theta}_e$ respectively be the enforced, valid and discarded technical constraints for e. The enforced technical constraints are the constraints implicitly or explicitly expressed during the drawing phase. The valid technical constraints regroup all the other technical constraints that could be valid at this point, given the restrictions of the enforced technical constraints that were presented in Section 8.3.1.

If a given component is mandatory, we cannot suggest it to be optional, and if it is single-valued, we cannot suggest it to be multivalued. Similarly, a textual attribute could be of any other type, while a real attribute could only also be of the type integer. Finally, if the component is optional, it could require every other optional components of the entity type.

Algorithm 8.1 formalises this initialisation process for the technical constraints. For the entity type `Product` associated with its homographic form, this could yield the following enforced constraints:

- for the attribute `Code` : (`Code`, `cardinality`, $(1, 1)$), (`Code`, `value type`, `text`), (`Code`, `value type`, `50`), (`Code`, `prerequisite`, $\varnothing$)
- for the attribute `Price` : (`Price`, `cardinality`, $(0, 1)$), (`Price`, `value type`, `real`), (`Price`, `value type`, `50`), (`Price`, `prerequisite`, $\varnothing$)
- for the role whose associated entity type is `Provider (primary)` : (`Provider (primary)`, `cardinality`, $(0, 1)$), (`Provider (primary)`, `value type`, $\varnothing$), (`Provider (primary)`, `value type`, $\varnothing$), (`Provider (primary)`, `prerequisite`, $\varnothing$)

It would also yield the following valid constraints suggestions:

- for the attribute `Code` : (`Code`, `value type`, `integer`), (`Code`, `value type`, `real`), (`Code`, `value type`, `boolean`), (`Code`, `value type`, `date`), (`Code`, `value size`, `0`)
- for the attribute `Price` : (`Price`, `cardinality`, $(1, 1)$), (`Price`, `value type`, `integer`), (`Price`, `value size`, `0`), (`Price`, `prerequisite`, {`Brand`, `Description`, `Provider (primary)`, `Provider (secondary)`})
- for the role whose associated entity type is `Provider (primary)` : (`Provider (primary)`, `cardinality`, $(1, 1)$), (`Provider (primary)`, `prerequisite`, {`Brand`, `Description`, `Price`, `Provider (secondary)`})

Existence constraints

Let Ξ_e, $\check{\Xi}_e$, $\bar{\Xi}_e$ respectively be the enforced, valid and discarded existence constraints for e. The enforced constraints are the constraints implicitly or explicitly expressed during the drawing phase, while the valid constraints regroup all the other constraints that could be valid at this point. This implies that any subset of optional components that isn't enforced and has more than one element could be subject to a existence constraint.

Algorithm 8.3 hence formalises the initialisation process for the existence constraints. For instance, if entity type `Special Good` of Fig. 7.14 had no initially enforced existence constraints, this could yield the following valid `coexistence` ones: ({`Conditions`, `Description`, `Price`}, `coexistence`), ({`Conditions`, `Description`}, `coexistence`), ({`Conditions`, `Price`}, `coexistence`), ({`Description`, `Price`}, `coexistence`).

Functional dependencies and Armstrong relations

Let $\mathfrak{F}_e$, $\check{\mathfrak{F}}_e$, $\bar{\mathfrak{F}}_e$ respectively be the enforced, valid and discarded functional dependencies for e. 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* (i.e. a relation that satisfies each FD implied by a given set of functional dependencies, but no functional dependency that is not implied by that set). Reaching such a state is obviously not trivial *per*

Algorithm 8.1 InitTechnicalConstraints : Initialise the technical constraints for the components of a given entity type (1/2)

Require: e is an entity type
Ensure: Θ_e, $\check{\Theta}_e$, $\bar{\Theta}_e$ are respectively the initially enforced, valid and discarded technical constraints for e

```
1: procedure InitTechnicalConstraints(e, Θ_e, Θ̌_e, Θ̄_e)
2:     Θ_e, Θ̌_e, Θ̄_e ← ∅
3:     C ← getComponents(e)
4:     C' ← getOptionalComponents(e)
5:     for all C ∈ C do
6:         mincard ← getMinimumCardinality(C)
7:         maxcard ← getMaximumCardinality(C)
8:         Θ_e ← Θ_e ∪ {(C, cardinality, (mincard, maxcard)}
9:         v ← ∅
10:        s ← ∅
11:        if getType(C) = SIMPLEATTRIBUTE then
12:            v ← getValueType(C)
13:            s ← getValueSize(C)
14:        end if
15:        Θ_e ← Θ_e ∪ {(C, value type, v)}
16:        Θ_e ← Θ_e ∪ {(C, value size, s)}
17:        r ← ∅
18:        if mincard = 0 then ▷ C is an optional component
19:            r ← getPrerequisiteComponents(C)
20:        end if
21:        Θ_e ← Θ_e ∪ {(C, prerequisite, r)}
22:        pC ← Dom(cardinality)\{(mincard, maxcard)}
23:        if mincard > 0 then
24:            pC ← pC\{(0, 1), (0, N)}
25:        end if
26:        if maxcard = 1 then
27:            pC ← pC\{(0, N), (1, N)}
28:        end if
29:        for all č_i ∈ pC do
30:            Θ̌_e ← Θ̌_e ∪ {(C, cardinality, č_i)}
31:        end for
32:        pV ← ∅
33:        if getType(C) = SIMPLEATTRIBUTE then
34:            if v = text then
35:                pV ← Dom(value type)\{text}
36:            else if v = real then
37:                pV ← {integer}
38:            end if
39:            Θ̌_e ← Θ̌_e ∪ {(C, value size, 0)}
40:        end if
```

▷ [continued on page 137]

Algorithm 8.2 InitTechnicalConstraints (2/2)

```
41:         for all ŏ_i ∈ pV do
42:             Ŏ_e ← Ŏ_e ∪ {(C, value type, ŏ_i)}
43:         end for
44:         ř ← ∅
45:         if mincard = 0 then
46:             ř ← {C_i ≠ C ∈ C'}
47:         end if
48:         Ŏ_e ← Ŏ_e ∪ {(C, prerequisite, ř)}
49:     end for
50: end procedure
```

Algorithm 8.3 InitExistenceConstraints : Initialise the existence constraints for a given entity type

Require: e is an entity type

Ensure: Ξ_e, $\check{\Xi}_e$, $\bar{\Xi}_e$ are respectively the initially enforced, valid and discarded existence constraints for e

```
1: procedure InitExistenceConstraints(e, Ξ_e, Ξ̌_e, Ξ̄_e)
2:     Ξ_e, Ξ̌_e, Ξ̄_e ← ∅
3:     C' ← getOptionalComponents(e)
4:     existenceconstraints ← getExistenceConstraints(e)
5:     for all existenceconstraint_i ∈ existenceconstraints do
6:         X_i ← getExistenceComponents(existenceconstraint_i)
7:         p_i ← getExistenceType(existenceconstraint_i)
8:         Ξ_e ← Ξ_e ∪ {(X_i, p_i)}
9:     end for
10:    for all X_i ⊆ C' with |X_i| > 1 do
11:        for all p_i ∈ 𝔼 do
12:            Ξ̌_e ← Ξ̌_e ∪ {(X_i, p_i)}
13:        end for
14:    end for
15:    Ξ̌_e ← Ξ̌_e \ Ξ_e
16: end procedure
```

se, and these principles are here inapplicable as a side effect 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 prefer to initialise a set of *high-level* possible dependencies, which would be the most general yet restrictive ones. These high-level dependencies claim that any component of a given entity type could determine the combined values of the other components. 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.

Algorithm 8.4 formalises the initialisation process for the technical constraints. For the entity type `Shop` associated with its homographic form in Fig. 7.14, this would yield the following top-level functional dependencies (considering that `Address` is the target entity type of the role played by `Shop`):

- for `Shop`: {`Shop Name`} → {`Telephone`, `Address`}, {`Telephone`} → {`Shop Name`, `Address`}, {`Address`} → {`Telephone`, `Shop Name`}.
- for the associated `Address`: {`Street`} → {`Zip code`, `City`}, {`Zip code`} → {`Street`, `City`}, {`City`} → {`Zip code`, `Street`}.

One might notice that we also consider optional components as possible members of the left-hand of functional dependencies. Indeed, we actually consider the `null` value (also noted ∅) as a value as such.

Algorithm 8.4 InitFunctionalDependencies : Initialise the functional dependencies for a given entity type

Require: e is an entity type
Ensure: $\mathfrak{F}_e$, $\check{\mathfrak{F}}_e$, $\bar{\mathfrak{F}}_e$ are respectively the initially enforced, valid and discarded functional dependencies for e

1: **procedure** InitFunctionalDependencies(e, $\mathfrak{F}_e$, $\check{\mathfrak{F}}_e$, $\bar{\mathfrak{F}}_e$)
2: $\quad\mathfrak{F}_e^{'} \leftarrow getFunctionalDependencies(e)$
3: $\quad$**if** $\mathfrak{F}_e^{'} \neq \emptyset$ **then**
4: $\quad\quad\mathfrak{F}_e \leftarrow \mathfrak{F}_e^{'}$
5: $\quad$**else**
6: $\quad\quad\mathfrak{F}_e \leftarrow \emptyset$
7: $\quad$**end if**
8: $\quad\mathcal{C} \leftarrow getComponents(e)$
9: $\quad\check{\mathfrak{F}}_e \leftarrow \{f : \{C\} \rightarrow \mathcal{C} \backslash \{C\} \mid C \in \mathcal{C}\}$
10: $\quad\bar{\mathfrak{F}}_e \leftarrow \emptyset$
11: **end procedure**

Unique constraints

Let Υ_e, $\check{\Upsilon}_e$ and $\bar{\Upsilon}_e$ respectively be the enforced, valid and discarded unique constraints for e. The enforced constraints are the constraints implicitly or explicitly expressed during the drawing phase, while the valid constraints should regroup all the other constraints that could be valid at this point. However, similarly to the functional dependencies, we prefer to start with *high-level* pos-

sible identifiers, i.e. single attributes that could identify the entity type by themselves.

Algorithm 8.5 formalises the initialisation process for the unique constraints. This would yield the following unique sets for entity type labelled `Service` in Fig. 7.14: {`Code`}, {`Description`}, {`Hourly rate`}.

For unique constraints, we also consider optional components as possible members of the valid identifiers. Indeed, we actually consider the `null` value (also noted $\varnothing$) as a value as such. We will discuss the implications of enforcing such a constraint later on.

Algorithm 8.5 InitUniqueConstraints : Initialise the unique constraints for a given entity type

Require: e is an entity type
Ensure: Υ_e, $\check{\Upsilon}_e$, $\bar{\Upsilon}_e$ are respectively the initially enforced, valid and discarded unique constraints for e

```
1: procedure InitUniqueConstraints(e, Υ_e, Ῠ_e, Ῡ_e)
2:     Υ_e, Ῠ_e, Ῡ_e ← ∅
3:     C ← getComponents(e)
4:     uniqueconstraints ← getUniqueConstraints(e)
5:     for all uniqueconstraint_i ∈ uniqueconstraints do
6:         X_i ← getUniqueComponents(uniqueconstraint_i)
7:         p_i ← getUniqueType(uniqueconstraint_i)
8:         Υ_e ← Υ_e ∪ {(X_i, p_i)}
9:     end for
10:    for all C_i ∈ C do
11:        for all p_i ∈ 𝕌 do
12:            Ῠ_e ← Ῠ_e ∪ {({C_i}, p_i)}
13:        end for
14:    end for
15:    Ῠ_e ← Ῠ_e \ Υ_e
16: end procedure
```

8.4.3 Analysing new data samples to suggest constraints and dependencies

Once the sets of constraints and dependencies have been initialised, we can take advantage of user input to acquire data samples that will progressively reduce the set of valid but unenforced constraints and dependencies. To be consistent with the previously enforced constraints and dependencies, any new tuple must respect the latter to be accepted.

Once a new tuple is acceptable, we can proceed with its analysis to determine which previously valid constraints and dependencies do not stand any

more. The invalidated constraints are discarded, while the invalidated functional dependencies are replaced by alternative dependencies.

Algorithm 8.6 formalises this process for the acquisition of a data sample for a form F and its associated set of entity types $\mathfrak{E}_F$. The updating process for each of the sets of constraints and dependencies are detailed subsequently.

Algorithm 8.6 AddDataSample : Add a data sample

Require: F is a form $\wedge$ $\mathfrak{E}_F$ is the set of entity types associated with F
Ensure: $\emptyset$

1: **procedure** AddDataSample($F, \mathfrak{E}_F$)
2: $d \leftarrow \varnothing$
3: **while** $d = \varnothing$ **do**
4: Ask> define: $d = \{(e_i, t_i) | (e_i \in \mathfrak{E}_F) \wedge (t_i \text{ is a tuple for } e_i)\}$
5: **if** d is a data sample for e **then**
6: $dataSampleIsValid \leftarrow$ `true`
7: **for all** $(e_i, t_i) \in d$ **do**
8: **if** $(\exists \theta \in \Theta_{e_i} : \mathcal{T}_{e_i} \cup \{t_i\} \not\models \theta) \vee (\exists \xi \in \Xi_{e_i} : \mathcal{T}_{e_i} \cup \{t_i\} \not\models \xi) \vee (\exists f \in \mathfrak{F}_{e_i} : \mathcal{T}_{e_i} \cup \{t_i\} \not\models f) \vee (\exists \upsilon \in \Upsilon_{e_i} : \mathcal{T}_{e_i} \cup \{t_i\} \not\models \upsilon)$ **then**
9: $dataSampleIsValid \leftarrow$ `false`
10: **end if**
11: **end for**
12: **if** $dataSampleIsValid =$ `true` **then**
13: **for all** $(e_i, t_i) \in d$ **do**
14: updateTechnicalConstraints($e_i, \mathcal{T}_{e_i}, t_i, \check{\Theta}_{e_i}, \bar{\Theta}_{e_i}$)
15: updateExistenceConstraints($e_i, \mathcal{T}_{e_i}, t_i, \check{\Xi}_{e_i}, \bar{\Xi}_{e_i}$)
16: updateFunctionalDependencies($e_i, \mathcal{T}_{e_i}, t_i, \mathfrak{F}_{e_i}, \check{\mathfrak{F}}_{e_i}, \bar{\mathfrak{F}}_{e_i}$)
17: updateUniqueConstraints($e_i, \mathcal{T}_{e_i}, t_i, \check{\Upsilon}_{e_i}, \bar{\Upsilon}_{e_i}, \mathfrak{F}_e, \check{\mathfrak{F}}_e$)
18: $\mathcal{T}_{e_i} \leftarrow \mathcal{T}_{e_i} \cup \{t_i\}$
19: **end for**
20: **else**
21: $d \leftarrow \varnothing$
22: **end if**
23: **else**
24: $d \leftarrow \varnothing$
25: **end if**
26: **end while**
27: **end procedure**

Technical constraints

Once we add a new tuple to the set of tuples associated with a given entity type, discarding the valid technical constraints that do not stand any more

is relatively straightforward, since it consists in removing the constraints with which the tuple does not agree.

Regarding the cardinalities, we remove the possible mandatory constraints for components that are empty, and possible single-valued constraints for components that are multivalued. For the attributes, 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. Finally, we remove all the require constraints for optional components if the suggested prerequisite components are not part of the non empty components of the tuple.

Algorithm 8.7 formalises this process. Consider for instance the initial constraints of Section8.4.2 and the data sample of Fig. 8.1. The tuple associated with the entity type `Product` invalidates the following constraints:

- for the attribute `Code` : (`Code`, `value type`, `integer`), (`Code`, `value type`, `real`), (`Code`, `value type`, `boolean`), (`Code`, `value type`, `date`), (`Code`, `value size`, `0`)
- for the attribute `Price` : (`Price`, `value size`, `0`), (`Price`, `prerequisite`, {`Brand`, `Description`, `Provider (primary)`, `Provider (secondary)`})
- for the role whose associated entity type is `Provider (primary)` : (`Provider (primary)`, `prerequisite`, {`Brand`, `Description`, `Price`, `Provider (secondary)`})

In return, it generated these alternative valid constraints:

- for the attribute `Code` : (`Code`, `value size`, `7`)
- for the attribute `Price` : (`Price`, `value size`, `3`), (`Price`, `prerequisite`, {`Brand`, `Description`, `Provider (primary)`})
- for the role whose associated entity type is `Provider (primary)` : (`Provider (primary)`, `prerequisite`, {`Brand`, `Description`, `Price`})

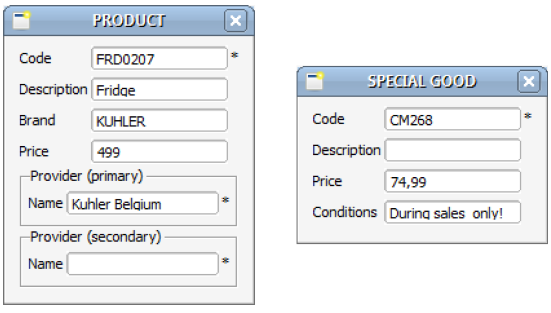


Figure 8.1: Data samples for the forms `Product` and `Special good`.

Algorithm 8.7 UpdateTechnicalConstraints : Update the technical constraints for a given entity type

Require: e is an entity type
$\wedge$ $\mathcal{T}_e$ is the current set of tuples associated with e
$\wedge$ t is a tuple to be added to $\mathcal{T}_e$
$\wedge$ $\check{\Theta}_e$, $\bar{\Theta}_e$ are the currently valid and discarded technical constraints for e
Ensure: $\check{\Theta}_e$, $\bar{\Theta}_e$ are the updated valid and discarded technical constraints for e and the set of tuples $\mathcal{T}_e \cup \{t\}$

procedure UpdateTechnicalConstraints($e, \mathcal{T}_e, t, \check{\Theta}_e, \bar{\Theta}_e$)
  $\mathcal{C} \leftarrow getComponents(e)$
  $\mathcal{C}' \leftarrow getOptionalComponents(e)$
  $\mathcal{C}'' \leftarrow \{C_i \in \mathcal{C}' | t[C_i] \neq \varnothing\}$
  $\Theta_e^{'} \leftarrow \emptyset$ ▷ The set of unsatisfied technical constraints
  for all $C_i \in \mathcal{C}$ **do**
    if $t[C_i] = \varnothing$ **then** ▷ C_i is optional and non initialised for t
      $\Theta_e^{'} \leftarrow \Theta_e^{'} \cup \{(C_i, \texttt{cardinality}, v_{i_j}) \in \check{\Theta}_e | v_{i_j} \in \{(1,1),(1,N)\}\}$
    else if $|t[C_i]| > 1$ **then** ▷ C_i is multivalued for t
      $\Theta_e^{'} \leftarrow \Theta_e^{'} \cup \{(C_i, \texttt{cardinality}, v_{i_j}) \in \check{\Theta}_e | v_{i_j} \in \{(0,1),(1,1)\}\}$
    end if
    if $getType(C_i) = \texttt{SIMPLEATTRIBUTE}$ **then**
      $V \leftarrow getPossibleValueTypes(t[C_i])$
      $\Theta_e^{'} \leftarrow \Theta_e^{'} \cup \{(C_i, \texttt{value type}, v_{i_j}) \in \check{\Theta}_e | v_{i_j} \notin V\}$
      $s \leftarrow getValueSize(t[C_i])$
      $S \leftarrow \{v_{i_j} | \exists (C_i, \texttt{value size}, v_{i_j}) \in \check{\Theta}_e\}$
      if $s > Max(S)$ **then**
        $\Theta_e^{'} \leftarrow \Theta_e^{'} \cup \{(C_i, \texttt{value size}, v_{i_j}) \in \check{\Theta}_e | v_{i_j} \in S\}$
        $\check{\Theta}_e \leftarrow \check{\Theta}_e \cup \{(C_i, \texttt{value size}, s)\}$
      end if
    end if
    if $t[C_i] \neq \varnothing$ **then**
      $\Theta_e^{'} \leftarrow \Theta_e^{'} \cup \{(C_i, \texttt{prerequisite}, v_{i_j}) \in \check{\Theta}_e | v_{i_j} \not\subseteq \mathcal{C}'' \backslash \{C_i\}\}$
    end if
  end for
  $\check{\Theta}_e \leftarrow \check{\Theta}_e \backslash \Theta_e^{'}$
  $\bar{\Theta}_e \leftarrow \bar{\Theta}_e \cup \Theta_e^{'}$
end procedure

Existence constraints

Once we add a new tuple to the set of tuples associated with a given entity type, discarding the valid 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, at most one 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.

Algorithm 8.8 formalises this process. Consider for instance the initial constraints of Section8.4.2 and the data sample of Fig. 8.1. The tuple associated with the entity type `Special good` invalidates:

- the following coexistence constraints: ({*Conditions*, *Description*, *Price*}, `coexistence`), ({*Conditions*, *Description*}, `coexistence`), ({*Description*, *Price*}, `coexistence`)
- the following exactly one constraints: ({*Conditions*, *Description*, *Price*}, `exactly one`), ({*Conditions*, *Price*}, `exactly one`)
- the following at most one constraints: ({*Conditions*, *Description*, *Price*}, `at most one`), ({*Conditions*, *Price*}, `at most one`)
- no least one constraints.

Algorithm 8.8 UpdateExistenceConstraints : Update the existence constraints for a given entity type

Require: e is an entity type
$\wedge$ $\mathcal{T}_e$ is the current set of tuples associated with e
$\wedge$ t is a tuple to be added to $\mathcal{T}_e$
$\wedge$ $\check{\Xi}_e$, $\bar{\Xi}_e$ are the currently valid and discarded existence constraints for e
Ensure: $\check{\Xi}_e$, $\bar{\Xi}_e$ are the updated valid and discarded existence constraints for e and the set of tuples $\mathcal{T}_e \cup \{t\}$

1: **procedure** UpdateExistenceConstraints($e, \mathcal{T}_e, t, \check{\Xi}_e, \bar{\Xi}_e$)
2: $\quad\mathcal{C}' \leftarrow getOptionalComponents(e)$
3: $\quad\mathcal{C}'' \leftarrow \{C_i \in \mathcal{C}' | t[C_i] \neq \varnothing\}$
4: $\quad$**for all** $\xi_i = (\mathcal{X}_i, p_i) \in \check{\Xi}_e$ **do**
5: $\quad\quad$**if** $(v_i = \texttt{coexistence} \wedge \mathcal{C}'' \neq \mathcal{C}' \wedge \mathcal{C}'' \neq \emptyset) \vee (v_i = \texttt{exactly one} \wedge |\mathcal{C}'' \cap \mathcal{X}_i| \neq 1) \vee (v_i = \texttt{at most one} \wedge |\mathcal{C}'' \cap \mathcal{X}_i| > 1) \vee (v_i = \texttt{at least one} \wedge |\mathcal{C}'' \cap \mathcal{X}_i| < 1)$ **then**
6: $\quad\quad\quad\check{\Xi}_e \leftarrow \check{\Xi}_e \backslash \{\xi_i\}$
7: $\quad\quad\quad\bar{\Xi}_e \leftarrow \bar{\Xi}_e \cup \{\xi_i\}$
8: $\quad\quad$**end if**
9: $\quad$**end for**
10: **end procedure**

Functional dependencies

When a new tuple is added, we analyse each valid functional dependency to check if there is an existing tuple of the tuple base that is conflictual, 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 is 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.

Algorithm 8.9 formalises this process. Consider for instance the initial functional dependencies of Section8.4.2 and the data samples of Fig. 8.2. If the set of tuples was initially empty, adding the first tuple doesn't jeopardise these dependencies. However, adding the second tuple invalidates the following dependencies:

- for `Shop`: {`Telephone`} → {`Shop Name`, `Address`}.
- for the associated `Address`: {`Zip code`} → {`Street`, `City`}, {`City`} → {`Zip code`, `Street`}.

In return, it yields the following alternative valid dependencies:

- for `Shop`:
  - {`Telephone`, `Shop Name`} → {`Address`}
  - {`Telephone`, `Address`} → {`Shop Name`}
- for the associated `Address`:
  - {`Street`, `Zip code`} → {`City`}
  - {`Street`, `City`} → {`Zip code`}
  - {`Zip code`} → {`City`}
  - {`City`} → {`Zip code`}

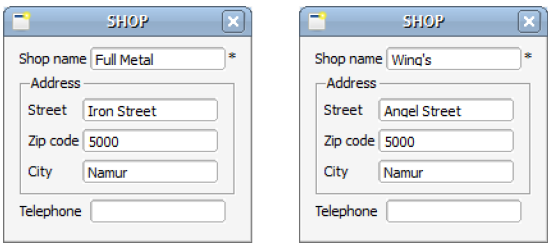


Figure 8.2: Data samples for the form `Shop`.

Algorithm 8.9 UpdateFunctionalDependencies : Update the functional dependencies for a given entity type

Require: e is an entity type
$\wedge$ $\mathcal{T}_e$ is the current set of tuples associated with e
$\wedge$ t is a tuple to be added to $\mathcal{T}_e$
$\wedge$ $\mathfrak{F}_e$, $\check{\mathfrak{F}}_e$, $\bar{\mathfrak{F}}_e$ are respectively the currently enforced, valid and discarded functional dependencies for e
Ensure: $\check{\mathfrak{F}}_e$, $\bar{\mathfrak{F}}_e$ are the updated valid and discarded functional dependencies for e and the set of tuples $\mathcal{T}_e \cup \{t\}$

1: **procedure** UpdateFunctionalDependencies($e, \mathcal{T}_e, t, \mathfrak{F}_e, \check{\mathfrak{F}}_e, \bar{\mathfrak{F}}_e$)
2: **for all** $f_i : \mathcal{L}_i \rightarrow \mathcal{R}_i \in \check{\mathfrak{F}}_e$ **do**
3: **if** $\exists t_j \in \mathcal{T}_e : t_j[\mathcal{L}_i] = t[\mathcal{L}_i] \wedge t_j[\mathcal{R}_i] \neq t[\mathcal{R}_i]$ **then**
4: $\check{\mathfrak{F}}_e \leftarrow \check{\mathfrak{F}}_e \backslash \{f_i\}$
5: $\bar{\mathfrak{F}}_e \leftarrow \bar{\mathfrak{F}}_e \cup \{f_i\}$
6: GenerateAlternatives($f_i, \mathfrak{F}_e, \check{\mathfrak{F}}_e, \bar{\mathfrak{F}}_e, \mathcal{T}_e \cup \{t\}$) ▷ See Alg. 8.10 on page 145
7: **end if**
8: **end for**
9: **end procedure**

Algorithm 8.10 GenerateAlternatives : Recursively generate alternatives "weaker" functional dependencies from the given one

Require: f is an unsatisfied functional dependency for a given entity type e
$\wedge$ $\mathcal{T}$ is a set of tuples for e
$\wedge$ $\mathfrak{F}$, $\check{\mathfrak{F}}$, $\bar{\mathfrak{F}}$ are respectively the currently enforced, valid and discarded functional dependencies for e
Ensure: $\mathfrak{F}$, $\check{\mathfrak{F}}$, $\bar{\mathfrak{F}}$ are respectively the enforced, valid and discarded functional dependencies for e that have been updated to include the alternatives for f

1: **procedure** GenerateAlternatives($f : \mathcal{L} \rightarrow \mathcal{R}, \mathfrak{F}, \check{\mathfrak{F}}, \bar{\mathfrak{F}}, \mathcal{T}$)
2: **if** $|\mathcal{R}| > 1$ **then**
3: **for all** $C \in \mathcal{R}$ **do**
4: GenerateAlternativeBranch($f1 : \mathcal{L} \rightarrow \mathcal{R}\backslash\{C\}, \mathfrak{F}, \check{\mathfrak{F}}, \bar{\mathfrak{F}}, \mathcal{T}$)
5: GenerateAlternativeBranch($f2 : \mathcal{L} \cup \{C\} \rightarrow \mathcal{R}\backslash\{C\}, \mathfrak{F}, \check{\mathfrak{F}}, \bar{\mathfrak{F}}, \mathcal{T}$)
6: ▷ See Alg. 8.11 on page 146
7: **end for**
8: **end if**
9: **end procedure**

Unique constraints

When adding a new tuple, that valid unique constraints can be easily verified by checking if an existing tuple already has the same values for the set of identifying components. Another option consists in taking advantage of the

Algorithm 8.11 GenerateAlternativeBranch : Tests a candidate functional dependency and generates alternatives if necessary

Require: f is a candidate functional dependency for a given entity type e
$\wedge$ $\mathcal{T}$ is a set of tuples for e
$\wedge$ $\mathfrak{F}$, $\check{\mathfrak{F}}$, $\bar{\mathfrak{F}}$ are respectively the currently enforced, valid and discarded functional dependencies for e
Ensure: $\mathfrak{F}$, $\check{\mathfrak{F}}$, $\bar{\mathfrak{F}}$ are respectively the enforced, valid and discarded functional dependencies for e that have been updated to include f and, if relevant, its alternatives

```
1: procedure GenerateAlternativeBranch(f, 𝔉, 𝔉̌, 𝔉̄, 𝒯)
2:     if f ∉ (𝔉 ∪ 𝔉̌ ∪ 𝔉̄) then ▷ f hasn't been tested yet
3:         if (𝒯 ⊨ f) ∧ (f ∉ 𝔉) then ▷ f is valid and has not been enforced yet
4:             𝔉̌ ← 𝔉̌ ∪ {f}
5:         else
6:             𝔉̄ ← 𝔉̄ ∪ {f}
7:             GenerateAlternative(f, 𝔉, 𝔉̌, 𝔉̄, 𝒯)
8:         end if
9:     end if
10: end procedure
```

fact that an enforced or valid functional dependency may induce an identifier for an entity type, if all the components of this entity type are mentioned in the left-hand or right-hand side of the functional dependency.

Algorithm 8.12 formalises this observation to update the sets of unique constraints. Consider for instance the initial constraints of Section8.4.2 and the data samples of Fig. 8.3 for the entity type `Special good`. If the set of tuples was initially empty, adding the first tuple doesn't jeopardise these dependencies. However, adding the second tuple invalidates the unique constraints associated with {`Description`}. In return, it yields the following alternative possible sets of identifiers: {`Code`, `Description`}, {`Description`, `Hourly rate`}.

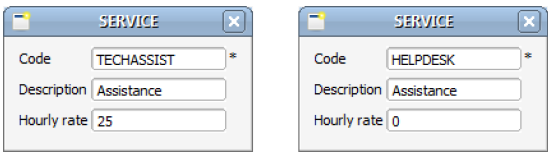


Figure 8.3: Data samples for the form `Service`.

Generating problematic tuples to help manage constraints and dependencies

Understanding the implications of a functional dependency is not always trivial and easy to grasp. Presenting the end-users with automatically generated data

Algorithm 8.12 UpdateUniqueConstraints : Update the unique constraints for a given entity type

Require: e is an entity type
$\wedge$ $\mathcal{T}_e$ is the current set of tuples associated with e
$\wedge$ t is a tuple to be added to $\mathcal{T}_e$
$\wedge$ $\check{\Upsilon}_e$, $\bar{\Upsilon}_e$ are respectively the currently valid and discarded unique constraints for e
Ensure: $\check{\Upsilon}_e$, $\bar{\Upsilon}_e$ are respectively the updated valid and discarded unique constraints for e and the set of tuples $\mathcal{T}_e \cup \{t\}$

1: **procedure** UpdateUniqueConstraints($e, \mathcal{T}_e, t, \check{\Upsilon}_e, \bar{\Upsilon}_e, \mathfrak{F}_e, \check{\mathfrak{F}}_e$)
2: $\mathcal{C} \leftarrow getComponents(e)$
3: $\mathcal{C}^* \leftarrow getMandatoryComponents(e)$
4: $\Upsilon_e^{'} \leftarrow \emptyset$ ▷ the new possibly valid identifiers
5: **for all** $f_i : \mathcal{L}_i \rightarrow \mathcal{R}_i \in (\mathfrak{F}_e \cup \check{\mathfrak{F}}_e)$ **do**
6: **if** $(\mathcal{L}_i \cup \mathcal{R}_i = \mathcal{C})$ **then**
7: $\Upsilon_e^{'} \leftarrow \Upsilon_e^{'} \cup \{(\mathcal{L}_i, \texttt{primary}), (\mathcal{L}_i, \texttt{secondary})\}$
8: **end if**
9: **end for**
10: $\check{\Upsilon}_e^{'} \leftarrow \check{\Upsilon}_e \cap \Upsilon_e^{'}$
11: $\bar{\Upsilon}_e \leftarrow \bar{\Upsilon}_e \cup (\check{\Upsilon}_e \backslash \check{\Upsilon}_e^{'})$
12: $\check{\Upsilon}_e \leftarrow \check{\Upsilon}_e^{'}$
13: **end procedure**

samples that would jeopardize the validity of existing functional dependencies could therefore help 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 $f : \mathcal{L} \rightarrow \mathcal{R}$ and the existing set of tuples $\mathcal{T}$ if: $\exists t' \in \mathcal{T} : t'[\mathcal{L}] = t[\mathcal{L}] \wedge t'[\mathcal{R}] \neq t[\mathcal{R}]$.

If we already have several tuples in the tuples set of a given entity type, we can therefore generate problematic tuples using Algorithm 8.13. For instance, the set of tuples built from the two data samples of Fig. 8.2 could yield the problematic data sample of Fig. 8.4 for the functional dependency $\{\texttt{Shop Name}\} \rightarrow \{\texttt{Telephone}, \texttt{Address}\}$.

Accepting the problematic data sample would imply discarding this functional dependency, but also consequently discard the following dependencies, for which alternatives must be generated:

- $\{\texttt{Address}\} \rightarrow \{\texttt{Shop Name}, \texttt{Telephone}\}$
- $\{\texttt{Telephone}, \texttt{Shop Name}\} \rightarrow \{\texttt{Address}\}$
- $\{\texttt{Telephone}, \texttt{Address}\} \rightarrow \{\texttt{Shop Name}\}$

Algorithm 8.13 GenerateProblematicTuple : Tests a candidate functional dependency and generates alternatives if necessary

Require: f is functional dependency for a given entity type e
$\wedge\ \mathcal{T} = \{t_1, t_2, ...\}$ is a set of tuples for e
Ensure: t is a problematic tuple for f or is $\varnothing$ it is was impossible to create it

```
1:  procedure GenerateProblematicTuple(f, 𝔉, 𝔉̌, 𝔉̄, 𝒯)
2:      t ← ∅
3:      i ← 1
4:      while i ≤ |𝒯| ∧ t = ∅ do
5:          j ← 1
6:          while j ≤ |𝒯| ∧ t = ∅ do
7:              if i ≠ j then
8:                  t ← t_i  ▷ Init the left components of the tuple
9:                  t[ℛ] ← t_j[ℛ]  ▷ Init the right components of the tuple
10:             end if
11:             if t ∈ 𝒯 then
12:                 t ← ∅
13:             end if
14:             j ← j + 1
15:         end while
16:         if t ∈ 𝒯 then
17:             t ← ∅
18:         end if
19:         i ← i + 1
20:     end while
21: end procedure
```

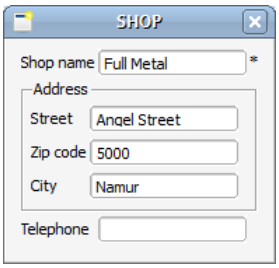


Figure 8.4: A problematic data sample for the form `Shop`.

Besides, this would impacts on the possible identifiers of `Shop`, and could have also impacted the existence constraints if the problematic tuple had been generated from other tuples (this would have obviously required a more populated set of tuples).

In order to propose original problematic tuples, it is appropriate to store

the discarded problematic tuples into a set $\bar{\mathcal{T}}_e$ for each entity type e.

8.4.4 Acquiring constraints and dependencies

Another way to take advantage of user input is to directly acquire enforced or discarded constraints and dependencies, whenever they are trivial and easy to express for the participants. An alternative is to invite the end-users to arbitrate the valid constraints and dependencies that could be suggested after the acquisition of multiple data samples.

Directly providing constraints and dependencies

The end-users should be able to directly specify enforced or discarded constraints and dependencies, even without looking at possible suggestions. To be accepted as enforced, a given constraint or dependency must be satisfied by the existing set of tuples associated with the considered entity type. On the other hand, it can be discarded as long as it still qualifies as a constraint or dependency for the given entity type.

Let δ be a candidate constraint or a dependency for the entity type e with the set of tuples $\mathcal{T}_e$. Let Δ_e and $\bar{\Delta}_e$ respectively be the sets of enforced and discarded constraints or dependencies of the same type than δ. Algorithm 8.14 formalises this acquisition process.

Algorithm 8.14 EnforceOrDiscardCandidateConstraint : Enforce or discard a candidate constraint or dependency

Require: δ is a constraint or dependency for a given entity type e
$\wedge$ $\mathcal{T}_e$ is a set of tuples for e
$\wedge$ Δ_e, $\bar{\Delta}_e$ are the sets of currently enforced and discarded constraints or dependencies of the same type than δ for e
$\wedge$ α is `true` if the constraint should be enforced, and `false` if it should be discarded
Ensure: Δ_e contains δ if it is satisfied by $\mathcal{T}_e$, and $\bar{\Delta}_e$ contains it otherwise

```
1: procedure EnforceOrDiscardCandidateConstraint(δ, 𝔗_e, Δ_e, Δ̄_e, α)
2:     if δ is a technical, existence, functional or unique constraint or dependency for e then
3:         if T_e ⊨ δ ∧ α = true then
4:             Δ_e ← Δ_e ∪ {δ}
5:         else
6:             Δ̄_e ← Δ̄_e ∪ {δ}
7:         end if
8:     end if
9: end procedure
```

Arbitrating valid constraints and dependencies

Alternatively, the participants can also take advantage of the valid constraints and dependencies to arbitrate them, i.e. to enforce 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 valid for the given entity type. Algorithm 8.15 formalises this acquisition process.

Algorithm 8.15 EnforceOrDiscardValidConstraint : Enforce or discard a candidate constraint or dependency

Require: $\delta \in \check{\Delta}_e$ is a valid and unenforced constraint or dependency for a given entity type e
$\wedge$ Δ_e, $\bar{\Delta}_e$ are the sets of currently enforced and discarded constraints or dependencies of the same type than δ for e
$\wedge$ α is `true` if the constraint should be enforced, and `false` if it should be discarded

Ensure: Δ_e contains δ if it is satisfied by $\mathcal{T}_e$, and $\bar{\Delta}_e$ contains it otherwise
$\wedge$ $\check{\Delta}_e$ does not contain δ anymore

```
1: procedure EnforceOrDiscardValidConstraint(δ, 𝔗_e, Δ_e, Δ̄_e, α)
2:     if α = true then
3:         Δ_e ← Δ_e ∪ {δ}
4:     else
5:         Δ̄_e ← Δ̄_e ∪ {δ}
6:     end if
7: end procedure
```

One can suspect that enforcing or discarding a constraint or a dependency may impact on the constraints or dependencies of other types. Such a synergy actually exists between functional dependencies and unique constraints. Indeed, discarding a valid functional dependency may change the valid unique constraints, whereas enforcing a unique constraints automatically enforces its underlying functional dependency. 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 approachable 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 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 valid 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 valid dependencies using Armstrong's axioms (recall Section 3.4.2). For a given entity type e, let us consider two different functional dependencies of $\tilde{\mathfrak{F}}_e$, namely $f_i : \mathcal{L}_i \rightarrow \mathcal{R}_i$ and $f_j : \mathcal{L}_j \rightarrow \mathcal{R}_j$. In particular, we will hide f_j if one of the following situations occurs:

1. $\mathcal{L}_i \subset \mathcal{L}_j \wedge \mathcal{R}_i = \mathcal{R}_j$
2. $\mathcal{L}_i = \mathcal{L}_j \wedge \mathcal{R}_i \supset \mathcal{R}_j$
3. $\exists \mathcal{W} \neq \emptyset : \mathcal{L}_i \cup \mathcal{W} = \mathcal{L}_j \wedge \mathcal{R}_i \cup \mathcal{W} = \mathcal{R}_j$
4. $\exists \mathcal{W} \neq \emptyset : \mathcal{L}_i \cup \mathcal{W} = \mathcal{L}_j \wedge \mathcal{R}_i = \mathcal{R}_j \cup \mathcal{W}$

In the first case, the left-hand side of f_i is smaller than the one of f_j and their right-hand sides are equal, which makes f_j redundant and gives f_i more weight in the balance. In the second case, f_j can be obtained from f_i by decomposition, which makes it redundant. In the third case, f_j can be obtained from f_i by augmentation. We can indeed observe that the left-hand sides and right-hand sides only differ by the same set of components. Finally, in the fourth case, we can decompose f_i into $\mathcal{L}_i \rightarrow \mathcal{W}$ and $\mathcal{L}_i \rightarrow \mathcal{R}_j$, from which $f_j : \mathcal{L}_i \cup \mathcal{W} \rightarrow \mathcal{R}_j$ can be deduced.

Hiding these functional dependencies does not mean discarding them. Indeed, they are still valid, and may eventually become visible again with the progressive arbitration of the other dependencies. Still, this filtering should help keeping the focus of the end-users.

Preventing stalemates

One of the major risks during this process is to gather conflictual or problematic constraints or dependencies that would lead to a stalemate. Consider for instance a set of component $\mathcal{X}$. Declaring that the components of $\mathcal{X}$ should coexist and simultaneously that there should be at least one of them creates a puzzling situation since one constraint cannot be satisfied without infringing the other.

Detection mechanisms can obviously be set to detect these kinds of situation. However, this also highlights once again the primordial role of the analyst in our approach, as he is the most suited person to notice and prevent such cases. He should therefore help the end-users to avoid them by guiding him into structuring their decisions in the most consistent fashion.

8.4.5 Editing the set of valid tuples and the sets of enforced constraints and dependencies

During this phase, the necessity to delete or edit a tuple $t \in \mathcal{T}_e$ for a given entity type e may occur. Similarly, an enforced FD may need to be edited or dismissed. Modifying these sets can be problematic, since they can create different types of conflicts.

Deleting and/or editing valid tuples

Deleting a tuple does not jeopardize the currently enforced and valid constraints and dependencies, but it could render previously discarded constraints and dependencies valid again. It is therefore important to recheck the latter, as formalised by Algorithm 8.16.

In contrast, editing a tuple can not contradict the currently enforced constraints and dependencies, and it can impact on the valid and discarded dependencies. Simply put, editing a tuple is actually similar to deleting and existing tuple then adding a new one.

Dismissing and/or editing enforced constraints and dependencies

Dismissing an enforced constraint or dependency δ for an entity type e simply comes down to moving it from its set Δ_e to $\breve{\Delta}_e$, since it is still satisfied by the current set of tuples $\mathcal{T}_e$.

However, editing an enforced constraint is more problematic, as it must still be satisfied by $\mathcal{T}_e$. This actually implies that editing a constraint or dependency is equivalent to discarding it, then choosing its replacement among the set of valid constraints or dependencies of the same type. The sole exception concerns the `value size` constraint for attributes, which can be edited as long as it doesn't conflict with the existing set of valid tuples.

8.4.6 Processing the end-users decisions

Whenever the participants are satisfied with their sets of tuples and enforced constraints and dependencies, we can proceed with the processing of their decisions to update the components of the pre-integrated schema s, as well as the form-based interfaces. This process should ideally occur once there is not more valid constraints or dependencies, i.e. they are all either enforced or discarded, but this is not mandatory. Let us now detail how each enforced type of constraints and dependencies are treated. Note that we only present the processes

Algorithm 8.16 RemoveTuple : Remove a valid tuple from the given set of tuples and update the constraints and dependencies accordingly.

Require: t is a valid tuple of $\mathcal{T}_e$
$\wedge$ $\mathcal{T}_e$ is a set of tuples for a given entity type e
Ensure: $\check{\Theta}_e$, $\bar{\Theta}_e$, $\check{\Xi}_e$, $\bar{\Xi}_e$, $\check{\mathfrak{F}}_e$, $\bar{\mathfrak{F}}_e$, $\check{\Upsilon}_e$, $\bar{\Upsilon}_e$ are updated to take in account the previously discarded constraints and dependencies that are now valid again

```
1: procedure RemoveTuple(t, T_e)
2:     T_e ← T_e \ t
3:     for all θ ∈ Θ̄_e do
4:         if T_e ⊨ θ then
5:             Θ̌_e ← Θ̌_e ∪ {θ}
6:             Θ̄_e ← Θ̄_e \ {θ}
7:         end if
8:     end for
9:     for all ξ ∈ Ξ̄_e do
10:        if T_e ⊨ ξ then
11:            Ξ̌_e ← Ξ̌_e ∪ {ξ}
12:            Ξ̄_e ← Ξ̄_e \ {ξ}
13:        end if
14:    end for
15:    for all f ∈ F̄_e do
16:        if T_e ⊨ f then
17:            F̌_e ← F̌_e ∪ {f}
18:            F̄_e ← F̄_e \ {f}
19:        end if
20:    end for
21:    for all υ ∈ Ῡ_e do
22:        if T_e ⊨ υ then
23:            Y̌_e ← Y̌_e ∪ {υ}
24:            Ῡ_e ← Ῡ_e \ {υ}
25:        end if
26:    end for
27: end procedure
```

for the entity types for the sake of brevity, but also because the updating of the forms is intuitively similar.

Technical constraints

Since the set of enhanced technical constraints contains one constraint per component and type of technical constraint, the easiest way to process the users decision is to treat them one by one to update the underlying data model accordingly. Algorithm 8.17 formalises this processing for the technical constraints.

Algorithm 8.17 ProcessTechnicalConstraints : Process the technical constraints for the components of a given entity type

Require: e is an entity type
$\wedge\ \Theta_e$ is the set of enforced technical constraints for e
Ensure: e is updated to take into account the enforced constraints

```
1: procedure ProcessTechnicalConstraints(e, Θ_e)
2:     C ← getComponents(e)
3:     A ← getAttributes(e)
4:     for all C ∈ C do
5:         (mincard, maxcard) ← v_i|(C, cardinality, (mincard, maxcard)) ∈ Θ_e
6:         setMinimumCardinality(C, mincard)
7:         setMaximumCardinality(C, maxcard)
8:         r ← v_i|(C, cardinality, v_i) ∈ Θ_e
9:         setPrerequisiteComponents(C, r)
10:        if C ∈ A then
11:            v ← v_i|(C, value type, v_i) ∈ Θ_e
12:            setValueType(C, v)
13:            s ← v_i|(C, value size, v_i) ∈ Θ_e
14:            setValueSize(C, s)
15:        end if
16:    end for
17: end procedure
```

Consider for instance that the technical constraints θ_1 = (`Customer Number`, `value type`, `integer`) for the entity type `Customer` and θ_2 = (`Vat Number`, `cardinality`, (1, 1)) for the entity type `Provider` (see Fig.7.14) are enforced. Processing them will practically modify the value type of the `Customer Number` and set `Provider` as a mandatory component.

Existence constraints

Let $\Xi_e^{'}$ be the set of existence constraints that were defined before the nurturing process, and that can be obtained using Algorithm 8.3. The first step consists in removing all the constraints of $\Xi_e^{'}$ that do not stand any more, then adding the constraints that were not included yet. Algorithm 8.18 formalises this process.

Consider for instance that the existence constraint ξ = ({`Zip Code`, `City`}, `at least one`) for the entity type `Address` (associated with a `Customer`, as shown in Fig.7.14) is enforced. Processing them will practically create an group containing the attributes `Zip Code` and `City` in the entity type `Address`, then constraint this group with an `at least one` predicate.

Algorithm 8.18 ProcessExistenceConstraints : Process the existence constraints for a given entity type

Require: e is an entity type
$\wedge$ Ξ_e is the current set of enforced existence constraints for e
$\wedge$ $\Xi_e^{'}$ is the initial set of enforced existence constraints for e
Ensure: e is updated to take into account the enforced constraints

1: **procedure** ProcessExistenceConstraints(e, Ξ_e, $\Xi_e^{'}$)
2: $\Xi_e^{''} \leftarrow \Xi_e \cap \Xi_e^{'}$ ▷ the initially enforced constraints that were not dismissed
3: **for all** $\xi_i = (\mathcal{X}_i, p_i) \in \Xi_e^{'} \backslash \Xi_e^{''}$ **do**
4: $removeExistenceConstraint(e, p_i, \mathcal{X}_i)$
5: **end for**
6: **for all** $\xi_i = (\mathcal{X}_i, p_i) \in \Xi_e \backslash \Xi_e^{''}$ **do**
7: $addExistenceConstraint(e, p_i, \mathcal{X}_i)$
8: **end for**
9: **end procedure**

Functional dependencies

Let $\mathcal{F}_e^{'}$ be the set of functional dependencies that were defined before the nurturing process, and that can be obtained using Algorithm 8.4. The first step consists in removing all the dependencies of $\mathcal{F}_e^{'}$ that do not stand any more, then adding the dependencies that were not included yet. Algorithm 8.19 formalises this process.

Algorithm 8.19 ProcessFunctionalDependencies : Process the functional dependencies for a given entity type

Require: e is an entity type
$\wedge$ $\mathfrak{F}_e$ is the set of enforced functional dependencies for e
$\wedge$ $\mathcal{F}_e^{'}$ is the initial set of enforced functional dependencies for e
Ensure: e is updated to take into account the enforced dependencies

1: **procedure** ProcessExistenceConstraints(e, $\mathcal{F}_e$, $\mathcal{F}_e^{'}$)
2: $\mathcal{F}_e^{''} \leftarrow \mathcal{F}_e \cap \mathcal{F}_e^{'}$ ▷ the initially enforced dependencies that were not dismissed
3: **for all** $f_i = (\mathcal{X}_i, p_i) \in \mathcal{F}_e^{'} \backslash \mathcal{F}_e^{''}$ **do**
4: $removeFunctionalDependency(e, f_i)$
5: **end for**
6: **for all** $f_i = (\mathcal{X}_i, p_i) \in \mathcal{F}_e \backslash \mathcal{F}_e^{''}$ **do**
7: $addFunctionalDependency(e, f_i)$
8: **end for**
9: **end procedure**

Unique constraints

Let $\Upsilon_e^{'}$ be the set of unique constraints that were defined before the nurturing process, and that can be obtained using Algorithm 8.5. The first step consists in removing all the constraints of $\Upsilon_e^{'}$ that do not stand any more, then adding the constraints that were not included yet. Algorithm 8.20 formalises this process.

Algorithm 8.20 ProcessUniqueConstraints : Process the unique constraints for a given entity type

Require: e is an entity type
$\wedge$ Υ_e is the current set of enforced unique constraints for e
$\wedge$ $\Upsilon_e^{'}$ is the initial set of enforced unique constraints for e
Ensure: e is updated to take into account the enforced constraints

1: **procedure** ProcessUniqueConstraints(e, Υ_e, $\Upsilon_e^{'}$)
2: $\Upsilon_e^{''} \leftarrow \Upsilon_e \cap \Upsilon_e^{'}$ ▷ the initially enforced constraints that were not dismissed
3: **for all** $\upsilon_i = (\mathcal{X}_i, p_i) \in \Upsilon_e^{'} \backslash \Upsilon_e^{''}$ **do**
4: $removeUniqueConstraint(e, p_i, \mathcal{X}_i)$
5: **end for**
6: **for all** $\upsilon_i = (\mathcal{X}_i, p_i) \in \Upsilon_e \backslash \Upsilon_e^{''}$ **do**
7: $addUniqueConstraint(e, p_i, \mathcal{X}_i)$
8: **end for**
9: **end procedure**

Consider for instance that the unique constraint $\upsilon = (\{\texttt{Vat number}\}, \texttt{primary})$ for the entity type `Provider` is enforced. Processing them will practically create an group containing the attribute `Vat number` in the entity type `Provider`, then set this group as a primary identifier for the entity type.

8.5 Output

At the end of this interactive process, the pre-integrated schema s has been augmented by all the constraints and dependencies that were enforced using the end-users input. At this point, it is necessary to observe that the constraints and dependencies that were not explicitly enforced have not been implicitly added. This underlines once again the importance of the analysts to guide the end-users appropriately.

After this process, the appearance of the forms is overall the same, the only visible modifications concerning the cardinality of the fields, and possible value type and size restrictions when inputting through the forms. The main modifications therefore concerns the underlying schema. Fig. 8.5 illustrates how our running example could have been nurtured. As we can see, there was

no trivial identifier for the different `Address` entity types, and that no one was mentioned for the entity type labelled `Orders` either.

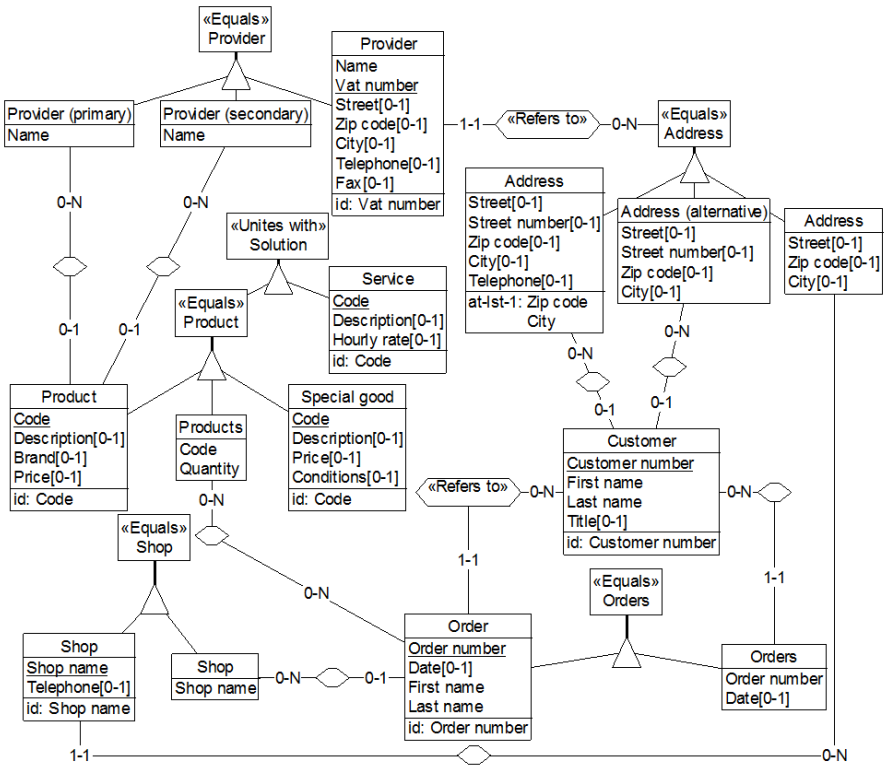


Figure 8.5: The pre-integrated schema of the example after the nurturing step.

Chapter 9

BIND

Completing the integration of the conceptual schema

In this chapter, we take advantage of the previous steps to complete the integration process of the different pieces of specification. Based on the pre-integrated schema obtained during the *Investigate* phase and the constraints and dependencies obtained through the *Nurture* phase, the semantically equivalent structures are arbitrated and integrated into a a non-redundant conceptual schema representing the data model conveyed by the user-drawn form-based interfaces.

We therefore first expose the elements that need to be arbitrated, then present our integration strategy. We subsequently explain how to manage the process, and present the resulting output.

9.1 Delimiting elements to integrate

As can be observed in Fig. 8.5, there are still different types of challenges to arbitrate at this point of the RAINBOW process. They notably concern the appropriate moving and integration of:

- the components of entity types involved in IS-A hierarchies that can be upwardly inherited;
- the components of entity types that are actually references to other entity types;
- the attributes of entity types would be better placed in relationship types involving these entity types;
- the constraints and dependencies involving these components.

As we have seen in Section 3.4.3, different transformational techniques exist to handle the integration of similar objects into non-redundant structures. Among these techniques, we choose to work with:

- *n-ary integration* for handling upward inheritance and solving the constraints, because of the potential multiple occurrences of key concepts;
- *binary integration* for referential components and attributes that need to be moved from entity types into relationship types.

All these transformations on the schema must maintain the traceability of elements, so that any widget of the user-drawn form-based interfaces can still found its counterpart in the data model.

9.2 Managing the process

One of the main challenges inherent to this process is to manage simultaneously all these transformation, since one does not really prevail on the others. We will hence present independently each type of sub process that needs to be led by the participants, and that could be interrupted to start another one. These sub processes all work around the same elements.

Let us consider $\mathcal{E}_s$, the set of data elements of the schema s, and $\mathcal{E} \in \mathcal{E}_s$, the set of entity types of s, which we want to integrate. Since we want to integrate elements of $\mathcal{E}_s$, some of them will be removed. We call $\mathcal{E}_s^\sharp$, the set that will receive those elements, so that we can still access their properties.

Let us also introduce the function $\phi : X \to Y$, that will contain the mapping between removed components of entity types and their associated "integrated" components, i.e. the component by which they are replaced. The set $X \subseteq \mathcal{E}_s^\sharp$ is the domain of ϕ (noted $Dom(\phi)$), and the set $Y \subseteq \mathcal{E}_s$ is the codomain of ϕ (noted $Codomain(\phi)$).

9.2.1 Arbitrating upward inheritance for IS-A relationships

In this step, we analyse the hierarchical organisation of entity types (declared as equal, specialised or united during the *Investigate* step) to discover and arbitrate components that could be upwardly inherited by supertypes. For this purpose, let us note $e_i \rhd e_j$ the fact that e_i has e_j as supertype.

Since we ensured that there is no hierarchical cycles in schema s, each IS-A tree has a root supertype, possibly intermediary entity types, and leaf entity types. The idea is therefore to start by the leaves and recursively arbitrate their components to decide which ones can be moved to the supertypes.

This process can be handled as a discussion between the end-users and the analysts regarding the definition of a "super" form containing all the information concerning a given concept.

Practically, for each level of a given hierarchy, the end-users must transparently define the sets of equivalent components that can be replaced by a single component into the supertype. Typically, this would concern attributes bearing the same terminology, or roles for relationship types involving the same entity type.

Consequently, a new component is created and the equivalent components are removed from the schema but stored in order to arbitrate their properties and constraints later on. Algorithm 9.1 formalises this process.

Algorithm 9.1 MoveInheritedComponents : arbitrate and move the relevant components from entity types to their supertypes

Require: $\mathfrak{E}$ is the set of entity types to be arbitrated
$\wedge\ \phi$ is the mapping function between components and their integrated counterpart
Ensure: $\mathfrak{E}$ is updated
$\wedge\ \phi$ is updated

1: **procedure** MoveInheritedComponents($\mathfrak{E}$)
2: $\hat{\mathfrak{E}} \leftarrow \{e_i \in \mathfrak{E} | (\exists e_j \in \mathfrak{E} : e_j \rhd e_i) \wedge (\nexists e_k \in \mathfrak{E} : e_i \rhd e_k)\}$
3: **for all** $e \in \hat{\mathfrak{E}}$ **do**
4: MoveInheritedComponentsRecursive($e, \mathfrak{E}, \phi$) ▷ See Algorithm 9.2 on page 162
5: **end for**
6: **end procedure**

Take for instance the hierarchy with `Solution` as root in Fig. 8.5. We start by confronting `Product`, `Products` and `Solution`, for which the user may decide to move and integrate the following components into the supertype `Product`:

- `Code` from `Product`, `Products` and `Special Good`
- `Description` from `Product` and `Special Good`
- `Brand` from `Product`
- `Price` from `Product` and `Special Good`
- the role leading to `Provider (primary)` from `Product`
- the role leading to `Provider (secondary)` from `Product`
- the role leading to `Order` from `Products`

Then, the supertype `Product` is confronted to `Service`, for which the user may decide to move and integrate the following components into the supertype `Solution`:

Algorithm 9.2 MoveInheritedComponentsRecursive : Recursively arbitrate and move the relevant components to the given entity type from its subtypes (1/2)

Require: e is the supertype for which the components must be arbitrated
$\wedge$ $\mathfrak{E}$ is a set of entity types containing the subtypes of e
$\wedge$ ϕ is the mapping function between components and their integrated counterpart
Ensure: e is updated with new components that represent the integration of arbitrated components from the subtypes
$\wedge$ ϕ is updated accordingly

1: **procedure** MoveInheritedComponentsRecursive($e, \mathfrak{E}, \phi$)
2: $\mathcal{E} \leftarrow \{e_i \in \mathfrak{E} \mid e_i \rhd e\}$
3: **for all** $e_i \in \mathcal{E}$ **do**
4: **if** $\exists e_j \in \mathcal{E} \mid e_j \rhd e_i$ **then**
5: MoveInheritedComponentsRecursive($e_i, \mathfrak{E}, \phi$)
6: **end if**
7: **end for**
8: **for all** $e_i \in \mathcal{E}$ **do**
9: $\mathcal{C}_{e_i} \leftarrow getComponents(e_i)$
10: **for all** $C_i \in \mathcal{C}_{e_i}$ **do**
11: ▷ Get the equivalent components among all subtypes of e
12: $\mathcal{C}_i \leftarrow (\{C_i\} \cup \{C_j \neq C_i \mid (\exists e_j \in \mathcal{E} : C_j \in getComponents(e_j)) \wedge (C_j \equiv C_i) \wedge (getType(C_j) = getType(C_i))\})$
13: ▷ Choose components to integrate (at most one per subtype)
14: Ask> define: $\mathcal{C}_i^\sharp \subseteq \mathcal{C}_i \mid \forall C_j, C_k \in \mathcal{C}_i^\sharp, \exists e_j, e_k \in \mathfrak{E} : C_j \in getComponents(e_j) \wedge C_k \in getComponents(e_k) \wedge e_j \neq e_k$
15: **if** $\mathcal{C}_i^\sharp \neq \emptyset$ **then**
16: $\mathcal{C}_e \leftarrow getComponents(e)$
17: ▷ Choose corresponding component in supertype, if any
18: Ask> define: $\hat{\mathcal{C}}_i \subseteq \mathcal{C}_e \mid (\hat{\mathcal{C}}_i = \emptyset) \vee (\exists! C \in \mathcal{C}_e | \hat{\mathcal{C}}_i = \{C\})$
19: ▷ Create a vessel that will be completely specified later on
20: **if** $getType(C_i) =$ `SIMPLEATTRIBUTE` **then**
21: $\mathfrak{T} \leftarrow \{t_k \mid \exists C_k \in \mathcal{C}_i^\sharp \cup \hat{\mathcal{C}}_i : getTerm(C_k) = t_k\}$
22: **if** $\nexists! t \mid \mathfrak{T} = \{t\}$ **then**
23: Ask> choose/redefine: t
24: **end if**
25: $C_i^\sharp \leftarrow createSimpleAttribute(e, t)$
26: **else**
27: $e_{C_i} \leftarrow getTarget(C_i)$
28: $R \leftarrow createRelationshipType(e, e_{C_i})$
29: $C_i^\sharp \leftarrow getRole(e, R)$
30: **end if**
31: ▷ Set the integrated components associated with the new component

▷ [continued on page 163]

Algorithm 9.3 MoveInheritedComponentsRecursive (2/2)

```
32:                 C_i^♮ ← {C_j ∈ Dom(φ) | φ(C_j) ∈ (C_i^♯ ∪ Ĉ_i)}
33:                 for all C_j ∈ (C_i^♯ ∪ Ĉ_i ∪ C_i^♮) do
34:                     (re)define: φ(C_j) ↦ C_i^♯
35:                     removeComponent(C_j)
36:                 end for
37:             end if
38:         end for
39:     end for
40: end procedure
```

- `Code` from `Product` and `Service`
- `Description` from `Product` and `Service`
- the role leading to `Order` from `Product`

Another example is the `Order` hierarchy. When `Order` is confronted with `Orders`, the user may decide to move and integrate the following components into the supertype:

- `Order number` from `Order` and `Orders`
- `Date` from `Order` and `Orders`
- the roles leading to `Customer` from `Order` and `Orders`

Once the analysis of a hierarchy is completed, it may appear (possibly after leading other sub processes) that some entity types of the hierarchy are left with one or less components. The analyst should therefore decide if such entity types should be maintained, for instance for the legibility of the schema, or if it could be transformed, for instance, into a boolean attribute in the subtype.

In our example, we could end up with an empty subtype `Product` and a `Seasonal good containing` only the attribute `Conditions`. The entity type `Product` could therefore be simply deleted (after updating ϕ so that ϕ(`Product`) produces its supertype), while `Seasonal good containing` could be maintained, or integrated with its supertype.

Since there are multiple ways to handle these transformations and refine, this process should be left at the discretion of the analyst. However, he should ensure to maintain the traceability of the elements by updating ϕ appropriately.

Another element that must be handled once the hierarchy is "stabilised" is the definition of its type: disjunction, totality or partition. Once again, this process is left at the discretion of the analyst.

9.2.2 Arbitrating referential components

In this step, we analyse the entity types that were declared as *referencing* others during the *Investigate* phase. For this purpose, we examine each of these entity types and select which components may be moved their referred counterpart. If the referential role was inherited, the selectable components may be chosen from the subtypes as well.

Once these components are selected, they are moved and integrated with their possible counterpart components in the destination entity type. Algorithm 9.4 formalises this process.

For instance, we observe that `Provider` refers to `Address`. The arbitration could imply moving and integrating `Street`, `Zip Code`, `City`, `Telephone` and `Fax` from `Provider` into `Address`. Similarly, `First Name` and `Last Name` could be moved and integrated from `Order` into `Customer`.

9.2.3 Dispatching attributes from entity types to relationship types

In this step, we analyse the entity types involved into relationship types, and for which attributes (or attributes of the subtypes) actually describe a property of the relationship between these entity types rather than a property of their current owner.

These attributes are therefore misplaced and should be moved to these relationship types. This can typically be the case for entity types originally associated with fieldsets or tables that aggregate information on a given concept as well as additional details on the relation between that concept and the concept associated with the parent widget.

Once these attributes are identified, they must be moved into the appropriate relationship type. Algorithm 9.5 formalises this process.

For instance, an order actually mentions a certain quantity for each product that it contains. This implies that `Quantity` is rather an attribute for the relationship type existing between a `Product` and an `Order` than solely a `Product`'s property, and should therefore be moved accordingly.

9.2.4 Solving constraints and dependencies for integrated components

Moving and integrating components also implies managing the constraints and dependencies in which they were involved, and hence completing their specification. The integrated objects may carry conflictual constraints, or even invalidate the constraints that were previously defined for (one of) their source components. We specifically consider two types of conflicts.

Algorithm 9.4 MoveReferentialComponents

Require: $\mathfrak{E}$ is a set of entity types to analyse
$\wedge$ ϕ is the mapping function between components and their integrated counterpart
Ensure: the referential components for each entity type of $\mathfrak{E}$ are moved and integrated in the referred entity types
$\wedge$ ϕ is updated accordingly

procedure MoveReferentialComponents($\mathfrak{E}, \phi$)
  for all $e_i \in \mathfrak{E}$ **do**
    $\mathcal{C}_{e_i} \leftarrow getComponents(e_i) \cup \{C_j \mid \exists e_j \in \mathfrak{E} : C_j \in getComponents(e_j) \wedge e_j \rhd e_i\}$
    $\mathcal{E}_i \leftarrow \{e_j \in \mathfrak{E} : e_i \overset{c}{\rightrightarrows} e_j\}$
    for all $e_j \in \mathcal{E}_i$ **do**
      $\mathcal{C}_{e_j} \leftarrow getComponents(e_j)$
      for all $C_i \in \mathcal{C}_{e_i}$ **do**
        $\mathcal{C}_i \leftarrow (\{C_i\} \cup \{C_j \neq C_i \in \mathcal{C}_{e_i} \mid C_j \equiv C_i\})$
        ▷ Choose components to integrate (at most one per entity type)
        Ask> define: $\mathcal{C}_i^\sharp \subseteq \mathcal{C}_i \mid \forall C_k, C_l \in \mathcal{C}_i^\sharp,\ \exists e_k, e_l \in \mathfrak{E} : C_k \in getComponents(e_k) \wedge C_l \in getComponents(e_l) \wedge e_k \neq e_l$
        if $\mathcal{C}_i^\sharp \neq \emptyset$ **then**
          $\mathcal{C}_{e_j} \leftarrow getComponents(e_j)$
          ▷ Choose corresponding component in referred entity type, if any
          Ask> define: $\hat{\mathcal{C}}_i \subseteq \mathcal{C}_{e_j} \mid (\hat{\mathcal{C}}_i = \emptyset) \vee (\exists! C \in \mathcal{C}_{e_j} | \hat{\mathcal{C}}_i = \{C\})$
          ▷ Create a vessel that will be completely specified later on
          if $getType(C_i) =$ `SIMPLEATTRIBUTE` **then**
            $\mathfrak{T} \leftarrow \{t_k \mid \exists C_k \in \mathcal{C}_i^\sharp \cup \hat{\mathcal{C}}_i : getTerm(C_k) = t_k\}$
            if $\nexists! t \mid \mathfrak{T} = \{t\}$ **then**
              Ask> choose/redefine: t
            end if
            $C_i^\sharp \leftarrow createSimpleAttribute(e_j, t)$
          else
            $e_{C_i} \leftarrow getTarget(C_i)$
            $R \leftarrow createRelationshipType(e_j, e_{C_i})$
            $C_i^\sharp \leftarrow getRole(e_j, R)$
          end if
          ▷ Set the integrated components associated with the new component
          $\mathcal{C}_i^\natural \leftarrow \{C_j \in Dom(\phi) \mid \phi(C_j) \in (\mathcal{C}_i^\sharp \cup \hat{\mathcal{C}}_i)\}$
          for all $C_j \in (\mathcal{C}_i^\sharp \cup \hat{\mathcal{C}}_i \cup \mathcal{C}_i^\natural)$ **do**
            (re)define: $\phi(C_j) \mapsto C_i^\sharp$
            $removeComponent(C_j)$
          end for
        end if
      end for
    end for
  end for
end procedure

Algorithm 9.5 MoveAttributesToRelationship

Require: $\mathfrak{E}$ is the set of entity types to be analysed
$\wedge\ \phi$ is the mapping function between components and their integrated counterpart
Ensure: the components of $\mathfrak{E}$ that needed to be moved into a relationship type were effectively moved
$\wedge\ \phi$ is updated accordingly

1: **procedure** MoveAttributesToRelationship($\mathfrak{E}, \phi$)
2: **for all** $e_i \in \mathfrak{E}$ **do**
3: $\mathcal{A}_{e_i} \leftarrow getSimpleAttributes(e_i) \cup \{A_j | \exists e_j \in \mathfrak{E} : A_j \in getSimpleAttributes(e_j) \wedge e_j \rhd e_i\}$
4: $\mathcal{R}_{e_i} \leftarrow getRoles(e_i)$
5: **for all** $r_j \in \mathcal{R}_{e_i}$ **do**
6: $R_j \leftarrow getRelationshipType(r_j)$
7: ▷ Choose attributes to move into R_j
8: Ask> define: $\mathcal{A}_j^\sharp \subseteq \mathcal{A}_{e_i}$
9: **if** $\mathcal{A}_j^\sharp \neq \emptyset$ **then**
10: $\hat{\mathcal{A}}_{R_j} \leftarrow getSimpleAttributes(R_j)$
11: ▷ Choose corresponding component in R_j, if any
12: Ask> define: $\hat{\mathcal{A}}_j \subseteq \mathcal{A}_{R_j} \mid (\hat{\mathcal{A}}_j = \emptyset) \vee (\exists! A \in \mathcal{A}_{R_j} | \hat{\mathcal{A}}_j = \{A\})$
13: ▷ Create a vessel that will be completely specified later on
14: $\mathfrak{T} \leftarrow \{t_k \mid \exists A_k \in \mathcal{A}_j^\sharp \cup \hat{\mathcal{A}}_j : getTerm(A_k) = t_k\}$
15: **if** $\nexists! t \mid \mathfrak{T} = \{t\}$ **then**
16: Ask> choose/redefine: t
17: **end if**
18: $A_j^\sharp \leftarrow createSimpleAttribute(R_j, t)$
19: ▷ Set the integrated components associated with the new component
20: $\mathcal{A}_j^\natural \leftarrow \{A_k \in Dom(\phi) \mid \phi(A_k) \in (\mathcal{A}_j^\sharp \cup \hat{\mathcal{A}}_j)\}$
21: **for all** $A_k \in (\mathcal{A}_j^\sharp \cup \hat{\mathcal{A}}_j \cup \mathcal{A}_j^\natural)$ **do**
22: (re)define: $\phi(A_k) \mapsto A_j^\sharp$
23: $removeComponent(A_k)$
24: **end for**
25: **end if**
26: **end for**
27: **end for**
28: **end procedure**

First of all, for technical constraints, a constraint becomes problematic if for the a given component, there are different values associated with the given property $p \in \mathbb{T}$. For instance, the `value type` could be either `real` or `integer`. In such a case, a unifying value must be chosen.

Secondly, for other types of constraints and dependencies, let $\mathcal{X}$ be the set of components originally associated with a constraint or dependency δ. The

set $\mathcal{X}$ can be partitioned into $\mathcal{X}_1 \subseteq \mathcal{E}_s$, the subset of components that were not updated, and $\mathcal{X}_2 \subseteq Dom(\phi)$, the set of updated components. The constraint δ will therefore become problematic if the components of $\mathcal{X}_1 \cup \phi(\mathcal{X}_2)$ do not belong to the same entity type, or if $\phi(\mathcal{X}_2)$ has less elements that $\mathcal{X}_2$. There is no trivial solution for "repairing" such a problem, and the original constraint must therefore be reconsidered.

Mechanisms for detecting potential conflicts could obviously be set up to to prevent problematic transformations on-the-fly. However, we rather focus on integrating non-problematic constraints and dependencies, and to detecting problematic ones for further manual analysis. Algorithm 9.6 formalises this process.

9.2.5 Manual modifications

In addition to these transformations, additional analysis and manipulations can naturally be performed on the schema s to improve it according to the subjectivity of the analyst. However, as long as the subsequent transformations do not jeopardize the semantics of the schema and the previously defined specifications, they do not require the input of the end-users. In such cases, the analysts should ensure to maintain the traceability of the elements by updating ϕ appropriately.

9.2.6 Updating the forms

The final step of this process concerns the update process of the form. We won't detail the process, but intuitively, for this purpose, we need to propagate the technical constraints of integrated components to their source widgets. The invalidated constraints and dependencies need to be removed, while the constraints and dependencies that were replaced remain valid and consistent with the current schema.

9.3 Output

At the end of this interactive process, the pre-integrated schema s has been progressively transformed into an integrated schema, where the constraints and dependencies previously defined for each entity type, as well as the relationships specified between entity types have been processed.

Accordingly to the user input, all the redundant elements and structure have been integrated, misplaced components moved to the appropriate owner, and the constraints and dependencies have been adapted accordingly. Traceability

Algorithm 9.6 SolveConstraints : Solves the constraints for which elements have been transformed

Require: $\mathfrak{E}$ is the set of entity types to be analysed
$\wedge\ \phi$ is the mapping function between components and their integrated counterpart
Ensure: $\Delta^\sharp$ is the set of constraints for which updated copies were made
$\wedge\ \bar{\Delta}$ is the set of constraints that could not be properly integrated

procedure SolveConstraints($\mathfrak{E}, \phi, \Delta^\sharp, \bar{\Delta}$)
  $\Delta^\sharp, \bar{\Delta} \leftarrow \emptyset$
  for all $e \in \mathfrak{E}$ **do**
    $\mathcal{C}_e \leftarrow getComponents(e)$
    for all $C_i \in (\mathcal{C}_e \cap Codomain(\phi))$ **do**
      $\Theta \leftarrow \{\theta_j \mid \exists e_j \in \mathfrak{E}, \theta_j = (C_j, p_j, v_j) \in \Theta_{e_j} : \phi(C_j) = C_i\} \backslash \Delta^\sharp$
      for all $p_j \in \mathbb{T}$ **do**
        $\mathcal{V} \leftarrow \{v_k \mid \exists \theta = (C_l, p_j, v_k) \in \Theta\}$
        if $\nexists ! v \mid \mathcal{V} = \{v\}$ **then**
          Ask> choose/redefine: v
        end if
        $\theta \leftarrow (C_i, p_j, v)$
        $addConstraint(e_i, \theta)$
      end for
      $\Delta^\sharp \leftarrow \Delta^\sharp \cup \Theta$
      $\Xi \leftarrow \{\xi_j \mid \exists e_j \in \mathfrak{E}, \xi_j = (\mathcal{X}_j, p_j) \in \Xi_{e_j} : C_i \in \phi(\mathcal{X}_j)\} \backslash \Delta^\sharp$
      $\mathfrak{F} \leftarrow \{f_j \mid \exists e_j \in \mathfrak{E}, f_j : \mathcal{L}_j \rightarrow \mathcal{R}_j \in \mathfrak{F}_{e_j} : C_i \in \phi(\mathcal{L}_j \cup \mathcal{R}_j)\} \backslash \Delta^\sharp$
      $\Upsilon \leftarrow \{\upsilon_j \mid \exists e_j \in \mathfrak{E}, \upsilon_j = (\mathcal{X}_j, p_j) \in \Upsilon_{e_j} : C_i \in \phi(\mathcal{X}_j)\} \backslash \Delta^\sharp$
      for all $\delta \in \Xi \cup \mathfrak{F} \cup \Upsilon$ **do**
        $\mathcal{X} \leftarrow getComponents(\delta)$
        $\mathcal{X}' \leftarrow \{C_i \mid (\exists C_j \in \mathcal{X} : \phi(C_j) = C_i) \vee (C_i \notin Dom(\phi) \wedge C_i \in \mathcal{X})\}$
        $\mathcal{E} \leftarrow \{e \in \mathfrak{E} \mid \exists C \in getComponents(e) : C \in \mathcal{X}'\}$
        if $(|\mathcal{X}| = |\mathcal{X}'|) \wedge (\exists ! e \mid \mathcal{E} = \{e\})$ **then**
          $\delta' \leftarrow \delta$
          $replaceComponents(\delta', \mathcal{X}, \mathcal{X}')$
          $addConstraint(e, \delta')$
          $deleteConstraint(\delta)$
          $\Delta^\sharp \leftarrow \Delta^\sharp \cup \{\delta\}$
        else
          $\bar{\Delta} \leftarrow \bar{\Delta} \cup \{\delta\}$
        end if
      end for
    end for
  end for
end procedure

between the elements of the form-based interfaces and the elements of the underlying schema is ensured thanks the mapping function ϕ.

After this process, the appearance of the forms is overall the same, the only visible modifications concerning the cardinality of the fields, and possible value type and size restrictions when inputting through the forms. The main modifications therefore concerns the underlying schema. Fig. 9.1 illustrates how the running example could have been updated.

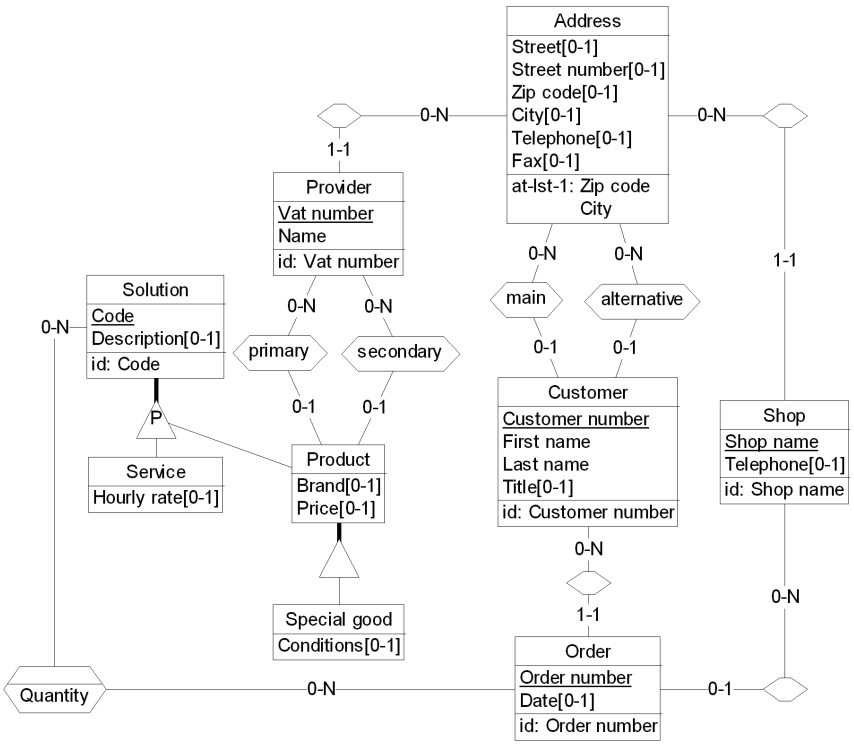


Figure 9.1: The schema of the running example after the binding phase.

Chapter 10

OBJECTIFY and WANDER

Generating and testing a playable prototype

In this chapter, we address the two last steps of the RAINBOW approach, which occur after the production of an integrated conceptual schema representing the underlying data model of the form-based interfaces that were drawn by the end-users. In order to ultimately validate the requirements conveyed by this schema, a prototypical application is generated and submitted to the end-users so they can test it.

Section 10.1 therefore presents the principles behind the generation and integration of applicative components, while Section 10.2 explains how to take advantage of the generated prototypical application as a means for validation. However, in this doctoral research, we mainly focused on the previous steps of the approach and only briefly expose these two steps, since the generation of the components is straightforward and the manipulation of a reactive prototype mainly adds another level of validation.

10.1 Objectify

The first stage of this ultimate validation consists in generating and integrating applicative components from the integrated conceptual schema. As we have seen, this process is relatively straightforward.

First of all, a database can be automatically generated using the transformational approach: the integrated conceptual schema is sequentially transformed into a logical schema, then a physical schema, and finally DDL code, from which

an operational database can be created using a compatible Database Management System (DBMS). CASE tools have proven very effective in supporting such a process.

Given the restrictions that were imposed in the RAINBOW approach, the automatic transformations that can be recursively applied on its conceptual structures are the following:

- entity types are transformed into tables
- monovalued simple attributes are transformed into columns
- multivalued simple attributes are transformed into entity types (representation by instance)
- relationship types with no attributes and that are not N to N are transformed into foreign keys
- other relationship types are transformed into entity types
- primary identifiers are transformed into primary keys
- secondary identifiers are transformed into uniques
- entity types with no identifiers receive a technical identifier
- domain, requires and existence constraints, as well as functional dependencies are transformed into check predicates
- IS-A relationships are transformed into relationship types with existence constraints, with respect to their type

Subsequently, access keys, spaces and clusters can be generated. Afterwards, if judged relevant by the participants, the database can be populated with the data samples provided by the end-users. Fig. 10.1 illustrates the physical schema that can be automatically obtained from the schema of Fig. 9.1, and Listing 10.1 shows an excerpt of the associated DDL code.

Listing 10.1: Excerpt of the DDL code generated from the schema of Fig. 10.1

```
create table Address (
     ID_address char(10) not null,
     Street varchar(50),
     Street_number varchar(50),
     Zip_code varchar(50),
     City varchar(50),
     Telephone varchar(50),
     Fax varchar(50),
     constraint ID_ID primary key (ID_address));

create table Customer (
     Customer_number numeric(50) not null,
     First_name varchar(50) not null,
     Last_name varchar(50) not null,
     Title varchar(50),
     ID_address_main char(10),
     ID_address_alternative char(10),
     constraint ID_Customer primary key (Customer_number));
```

```
alter table Address add constraint LSTONE_Address
     check(Zip_code is not null or City is not null);

alter table Customer add constraint FKmain
     foreign key (ID_address_main)
     references Address;

alter table Customer add constraint FKalternative
     foreign key (ID_address_alternative)
     references Address;

create unique index ID_IND on Address (ID_address);

create unique index ID_Customer on Customer (Customer_number);

create index FKmain on Customer (ID_address_main);

create index FKalternative on Customer (ID_address_alternative);
```

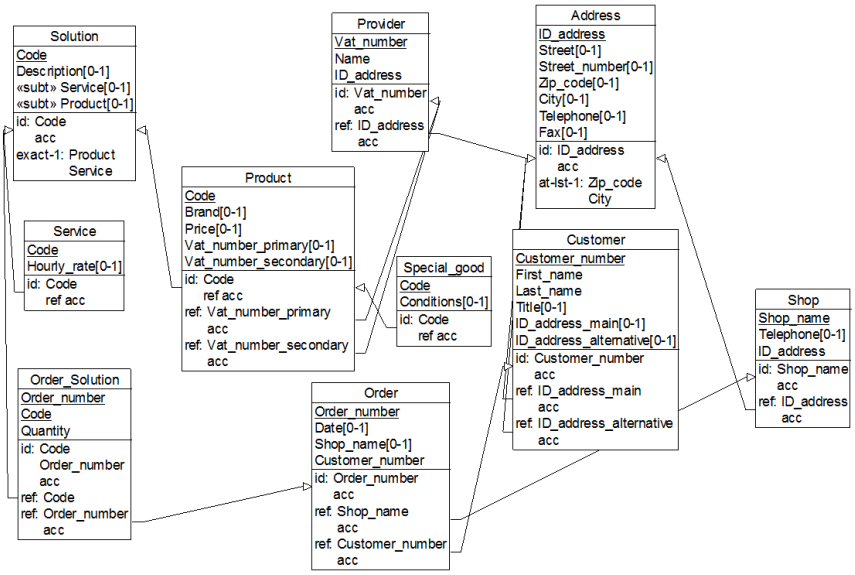


Figure 10.1: A possible physical schema for the running example.

Once the database has been set up in the DBMS, simple queries SQL to *select*, *insert*, *update* and *delete* rows of each table can be automatically generated. These queries can subsequently be connected to the form-based interfaces drawn by the end-users in order to make them reactive and report the messages of the database. The issue of querying through user interfaces has notably been studied in the Natural Forms Query Language (NFQL) [Embley, 1989] and the Guava framework [Terwilliger et al., 2006].

The final step consists in grouping the user-drawn interfaces in an operational environment. This implies creating the mechanisms for a central application granting navigational access to the forms and between them. The prototypical application thus created should enable to perform simple consulting and editing actions the database though the form-based interfaces, which would qualify it as a lightweight data manager for the database.

10.2 Wander

Finally, the last stage consists in confronting the end-users with the prototypical application to see if the static data requirements that were materialised correspond to their needs, which should ultimately validate the integrated conceptual schema.

The role of the analyst during this process is therefore to assist the users in the validation of the model through the use of the prototype, and to record their positive and negative remarks. Reporting mechanisms could also be integrated to the reactive form-based interfaces to gather these comments.

The evaluation of the elicited requirements through the manipulation of the associated lightweight data manager should eventually lead to end the requirements elicitation process or to loop back to the previous steps to add, delete or modify the specifications that were expressed.

Chapter 11

The RAINBOW Toolkit

A proof-of-concept tool support for the RAINBOW approach

In this chapter, we introduce the proof-of-concept CASE tool that was developed in order to support and experiment the RAINBOW approach. We consequently present the *RAINBOW Toolkit* that handles most of the processes developed in the five first steps of the approach. First, we present the overall design principles and the technologies that were used, then for each of the five steps, we present how the theoretical principles are instantiated.

11.1 Design principles

The RAINBOW Toolkit has been designed to support the processes of the five first steps of our approach, which are the most crucial ones. We therefore needed to develop an integrated environment that would allow to draw form-based interfaces and transparently manage their underlying data models.

For this purpose, we chose to use *Java* [Java, 2010], which is one of the most widely used multi-platform object-oriented programming languages, because it gave access to rich libraries serving our purpose.

Among them, we notably used the *Qt Jambi* library to develop the graphical interface of the application and manage the drawing of the forms. This library is the Java version of the Qt toolkit [Qt, 2010], which is a free and open-source cross-platform application development framework using C++.

Java also enables us to interact with the previously mentioned *DB-Main* CASE tool [DB-MAIN, 2010], through their Java API which gives access to

the DB-Main repository and GER constructs.

11.1.1 Available processes

To simplify the development, we organised the RAINBOW steps into exclusive and sequential objectives:

1. *Represent*: the end-users can draw and specify a set of form-based interfaces to perform usual tasks of their application domain. Once they are satisfied with their interfaces, the set of interfaces are automatically *adapted* into their underlying conceptual counterparts;
2. *Investigate*: the sets of terminological and structural ambiguities are consecutively computed and presented for arbitration. Based on the provided decisions, the pre-integrated schema is automatically built;
3. *Nurture*: using the interfaces that they drew, the end-users are invited to provide data examples as well as constraints and dependencies for each form. Each new data sample is analysed to adapt the suggested constraints and dependencies, and conversely, each new enforced constraint or dependency directly restricts the new data samples that can be encoded. The pre-integrated schema is updated accordingly;
4. *Bind*: the elements to integrate are computed and presented to the users for arbitration. Based on the provided decisions, the pre-integrated schema is transformed into a non redundant integrated conceptual schema.

The decisions made during each step are stored so that if required, the users can loop back to change a given decision then replay the other steps without requiring any re-arbitration for decisions that are still valid.

11.1.2 Implementation structure

The toolkit was implemented around the following main packages:

- The main `rainbow` package contains the main application, as well as the definitions of transversal constants and settings;
- The `rainbow.graphical.components` package (re)defines graphical components and widgets for the rendering the main application, such as dedicated dialog or message boxes;
- The `rainbow.graphical.rendering` package implements all the widgets for the rendering and manipulation of the form-based user interfaces that can be drawn;

- The `rainbow.project.components` package defines the main components of a RAINBOW project (such as a centralised Project manager, a Thesaurus, an XML handler), as well as utility classes;
- The `rainbow.toolboxes` package defines a set of classes to interact with the DB-Main repository, including a centralised class to manipulate DB-Main schemas (one per main process of the RAINBOW approach), the implementation of standard "black box" transformations, the definition of "virtual" elements to emulate DB-Main elements, as well as a converter between elements of the form-based interfaces and the DB-Main elements;
- The `rainbow.represent` package implements all the necessary classes to convey the specifications of the form-based interfaces, as well as dialogs to edit them;
- The `rainbow.adapt` package provides the mechanisms to transform a set of form-based interfaces into consecutive raw and refined schemas;
- The `rainbow.investigate.semantics` package provides term analysis on a schema based on orthographical and ontological comparison, the interactive arbitration of terminological similarities, as well as the processing of subsequent decisions;
- The `rainbow.investigate.syntax` package provides structural analysis on a schema based on patterns, the interactive arbitration of structural similarities, as well as the processing of subsequent decisions;
- The `rainbow.nurture` package enables to provide data samples, constraints and dependencies and to update the associated schema accordingly;
- The `rainbow.bind` package manages the integration process of the elements of a schema.

See Appendix C for the Java source code of the toolkit and its associated documentation.

11.2 Drawing and specifying form-based interfaces

11.2.1 Drawing environment

The toolkit provides access to all the elements specified in Chapter 5, as illustrated in Fig.11.1. For beginners, the environment is voluntarily pared down in order not to be overwhelming. However, advanced users may access additional docking windows with the interfaces list, the thesaurus, and the properties of the currently selected element.

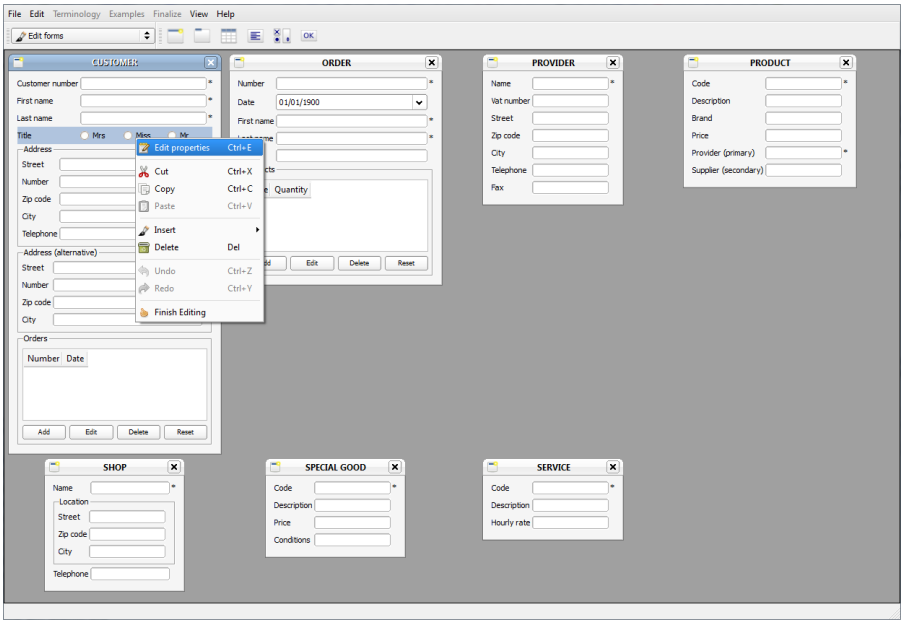


Figure 11.1: The RAINBOW toolkit's drawing environment.

The user starts by creating a form, before progressively inserting more elements into it. The layout of forms is voluntarily limited to a vertical sequence of elements, in order to keep the end-users focused on the *content* rather than the *form*. Each new widget is initialised with default values, so that the end-users can focus on the main properties (see Fig. 11.2), which are the label, the qualifier, the description and the cardinality for simple widgets, as well as the distinctiveness for containers. Advanced users may directly specify alternative terms, as well as technical or existence constraints.

Note that the toolkit restrict the edition of the form-based interfaces only to this drawing step. The next steps allow to update the terminology of the forms indirectly, but does not allow to change their structure. However, it is possible to loop back in the process at any time to edit the interfaces and replay the following steps without losing any of the intermediary decisions.

11.2.2 Suggesting terms on-the-fly

In order to reduce the semantic redundancies upstream, we provide an on-the-fly terminology suggester and analyser. When inserting a new widget or editing an existing one, the suggester automatically proposes possible terms,

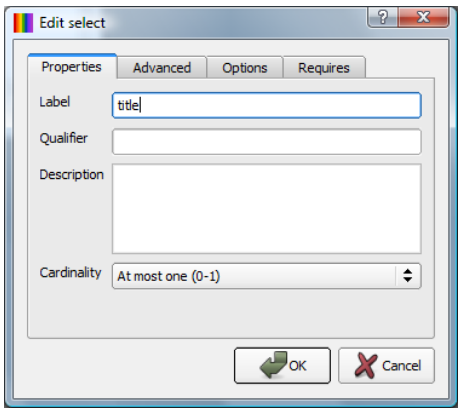


Figure 11.2: Editing a widget.

labels and qualifiers based on the existing terminology. If the user chooses to provide his own term, label and qualifier, the analyser compares them to the existing terminology to detect possible orthographical and/or ontological similarities, then asks the user for direct arbitration.

The conflictual siblings and non siblings are therefore presented, and the user may choose:

- to keep his own terminology, and possibly impose it to a selected set of other similar elements;
- to unify his own terminology, based on one of the similar elements;
- use a completely different terminology, which could in turn generate other similarities.

As we mentioned, we use *Jaro-Winkler*'s distance [Winkler, 1990] for lexical comparison, and the *WordNet* orthographical reference system[Fellbaum, 1998] for this purpose. Several java libraries have been developed to implement and manage string distance metrics, among which SimMetrics [Chapman, 2007], LingPipe [LingPipe, 2010] and SecondString [Carnegie Mellon University, 2006]. For our purpose, they are relatively equivalent, so we chose to work with the latter. Similarly, several java libraries have also been developed to interact with the WordNet database, among which JAWS [Spell, 2009], JWNL [Walenz and Didion, 2008] and MIT's JWI [Finlayson, 2009]. For our purpose, they are also relatively equivalent, so we chose to work with the latter.

11.2.3 Storing and adapting the interfaces

Once the users are satisfied with their set of form-based interfaces, the latter are stored and automatically adapted. We use XML [W3C, 2010] to store the initial interface drawings. Based on the specifications of Section 5.2, we defined a *Document Type Definition* (DTD) for our Simplified Form Model, as shown in Listing 11.1. Listing 11.2 shows an excerpt of the XML file associated with our running example.

Listing 11.1: The DTD specification for the RSFM

```
<!ELEMENT rtk (forms, dbmain?) >
   <!ATTLIST rtk label CDATA #IMPLIED >
   <!ATTLIST rtk description CDATA #IMPLIED >
<!ELEMENT forms (form+) >
   <!ATTLIST forms id CDATA #REQUIRED >
<!ELEMENT form ((fieldset|table|input|select|button)*, unique*, existence*) >
   <!ATTLIST form id CDATA #REQUIRED >
   <!ATTLIST form label CDATA #REQUIRED >
   <!ATTLIST form qualifier CDATA #IMPLIED >
   <!ATTLIST form description CDATA #IMPLIED >
   <!ATTLIST form term CDATA #IMPLIED >
<!ELEMENT fieldset ((fieldset|table|input|select|button)*, unique*, existence*, requires
     ?) >
   <!ATTLIST fieldset id CDATA #REQUIRED >
   <!ATTLIST fieldset label CDATA #REQUIRED >
   <!ATTLIST fieldset qualifier CDATA #IMPLIED >
   <!ATTLIST fieldset description CDATA #IMPLIED >
   <!ATTLIST fieldset term CDATA #IMPLIED >
   <!ATTLIST fieldset maxcard CDATA #FIXED "1" >
   <!ATTLIST fieldset mincard (0|1) "0" >
   <!ATTLIST fieldset transformbyinstance (false|true) #IMPLIED >
<!ELEMENT table ((input|select|button)*, unique*, existence*, requires?)>
   <!ATTLIST table id CDATA #REQUIRED >
   <!ATTLIST table label CDATA #REQUIRED >
   <!ATTLIST table qualifier CDATA #IMPLIED >
   <!ATTLIST table description CDATA #IMPLIED >
   <!ATTLIST table term CDATA #IMPLIED >
   <!ATTLIST table maxcard CDATA #FIXED "infinite" >
   <!ATTLIST table mincard (0|1) "0" >
   <!ATTLIST table transformbyinstance (false|true) #IMPLIED >
<!ELEMENT input (requires?) >
   <!ATTLIST input id CDATA #REQUIRED >
   <!ATTLIST input label CDATA #REQUIRED >
   <!ATTLIST input qualifier CDATA #IMPLIED >
   <!ATTLIST input description CDATA #IMPLIED >
   <!ATTLIST input term CDATA #IMPLIED >
   <!ATTLIST input mincard (0|1) "0" >
   <!ATTLIST input maxcard CDATA #FIXED "1" >
   <!ATTLIST input valuetype (boolean|date|datetime|integer|real|text) "text" >
   <!ATTLIST input valuesize CDATA "50" >
   <!ATTLIST input formula CDATA #IMPLIED >
<!ELEMENT select (option+, requires?) >
   <!ATTLIST select id CDATA #REQUIRED >
   <!ATTLIST select label CDATA #REQUIRED >
   <!ATTLIST select qualifier CDATA #IMPLIED >
   <!ATTLIST select description CDATA #IMPLIED >
   <!ATTLIST select term CDATA #IMPLIED >
   <!ATTLIST select mincard (0|1) "0" >
   <!ATTLIST select maxcard (1|infinite) "1" >
   <!ATTLIST select valuetype (boolean|date|integer|real|text) "text" >
   <!ATTLIST select valuesize CDATA "50" >
   <!ATTLIST select iseditable (false|true) "false" >
```

```
   <!ATTLIST select formula CDATA #IMPLIED >
<!ELEMENT option EMPTY >
   <!ATTLIST option id CDATA #REQUIRED >
   <!ATTLIST option label CDATA #REQUIRED >
   <!ATTLIST option qualifier CDATA #IMPLIED >
   <!ATTLIST option description CDATA #IMPLIED >
<!ELEMENT button (action+, requires?) >
   <!ATTLIST button id CDATA #REQUIRED >
   <!ATTLIST button label CDATA #REQUIRED >
   <!ATTLIST button qualifier CDATA #IMPLIED >
   <!ATTLIST button description CDATA #IMPLIED >
   <!ATTLIST button term CDATA #IMPLIED >
   <!ATTLIST button label CDATA #REQUIRED >
<!ELEMENT action (item*) >
   <!ATTLIST action label CDATA #REQUIRED >
   <!ATTLIST action description CDATA #REQUIRED >
<!ELEMENT unique (item+) >
   <!ATTLIST unique type (primary|secondary) "secondary" >
<!ELEMENT existence (item+) >
   <!ATTLIST existence type (coex|atmost1|atleast1|exactly1) "coex" >
<!ELEMENT requires (item+) >
<!ELEMENT item EMPTY>
   <!ATTLIST item refid CDATA #REQUIRED >
<!ELEMENT dbmain (schemas) >
   <!ATTLIST dbmain filename CDATA #REQUIRED >
<!ELEMENT schemas (schema*) >
<!ELEMENT schema EMPTY >
   <!ATTLIST schema class (AdaptRaw|AdaptRefined|InvestigateSemantics|InvestigateSyntax|
        Nurture|Bind|Objectify|Wander) "AdaptRaw" >
   <!ATTLIST schema iddbm CDATA #REQUIRED >
```

As can be observed, the DTD maintains a mapping between each step of the approach and a schema of the DB-Main repository. Therefore, when the edition of the forms is finished, a *raw* schema and a *refined* schema are consecutively created in the DB-Main repository using the mapping rules of Chapter 6, and the mapping is updated accordingly.

11.3 Arbitrating terminological and structural similarities

11.3.1 Terminological similarities

Once the refined schema has been produced, it is copied into a new schema to perform the terminological analysis. The schema is analysed to compute the sets of terminologically similar elements, using the principles of Section 7.1.2. As for the on-the-fly term suggester, we use the SecondString library to handle orthographical comparison with *Jaro-Winkler*'s distance, and JWI to interact with WordNet.

The sets that are discovered are compared with the previously arbitrated similar subsets, which are stored in the DB-Main repository using meta properties, so that valid pre-existing decisions are maintained. The sets are subsequently presented for arbitration, as illustrated in Fig. 11.3.

The elements bearing conflictual terms are grouped separately from the elements that bear non conflictual identical terms. For each set, subsets can be

Listing 11.2: The XML code associated with the `Customer` form of Fig. 5.10

```
<form description="" id="1" label="CUSTOMER" maxcard="1" mincard="1" qualifier="" term="
     CUSTOMER">
<input description="" id="2" label="Customer Number" maxcard="1" mincard="1" qualifier="
     " valuesize="" valuetype="integer"/>
<input description="" id="3" label="first name" maxcard="1" mincard="1" qualifier=""
     valuesize="" valuetype="text"/>
<input description="" id="4" label="Last Name" maxcard="1" mincard="1" qualifier=""
     valuesize="" valuetype="text"/>
<select description="" formula="" id="5" iseditable="false" label="title" maxcard="1"
     mincard="0" qualifier="" term="title" valuesize="50" valuetype="text">
<option description="" id="6" label="Mrs" qualifier=""/>
<option description="" id="7" label="Miss" qualifier=""/>
<option description="" id="8" label="Mr" qualifier=""/>
</select>
<fieldset description="" id="9" label="address" maxcard="1" mincard="0" qualifier=""
     term="address" transformbyinstance="false">
<input id="10" label="street" maxcard="1" mincard="0" valuesize="50" valuetype="text"/>
<input id="11" label="Number" maxcard="1" mincard="0" valuesize="50" valuetype="text"/>
<input id="12" label="Zip Code" maxcard="1" mincard="0" valuesize="50" valuetype="text"/
     >
<input id="13" label="city" maxcard="1" mincard="0" qualifier="" valuesize="50"
     valuetype="text"/>
<input description="" formula="" id="14" label="Telephone" maxcard="1" mincard="0"
     qualifier="" valuesize="50" valuetype="text"/>
</fieldset>
<fieldset description="" id="15" label="address" maxcard="1" mincard="0" qualifier="
     alternative" term="address" transformbyinstance="false">
<input id="16" label="street" maxcard="1" mincard="0" qualifier="" valuesize="50"
     valuetype="text"/>
<input id="17" label="number" maxcard="1" mincard="0" qualifier="" valuesize="50"
     valuetype="text"/>
<input id="18" label="Zip Code" maxcard="1" mincard="0" valuesize="50" valuetype="text"/
     >
<input id="19" label="city" maxcard="1" mincard="0" qualifier="" valuesize="50"
     valuetype="text"/>
</fieldset>
<table description="" id="20" label="orders" maxcard="infinite" mincard="0" qualifier=""
      transformbyinstance="true">
<input description="" formula="" id="21" label="number" maxcard="1" mincard="1"
     qualifier="" valuesize="50" valuetype="text"/>
<input description="" formula="" id="22" label="date" maxcard="1" mincard="0" qualifier=
     "" valuesize="" valuetype="date"/>
</table>
</form>
```

created and assign a unifying term. The conflictual elements are highlighted in form that contains them, thanks to the mapping between the data elements and the interface widgets.

Based on the provided decisions, the schema and the forms are updated with the new terminology, and the new decisions are stored in the DB-Main repository using meta properties.

11.3.2 Structural similarities

The terminologically updated schema is then copied into a new schema to perform the structural analysis. The schema is analysed to compute the sets of structurally similar elements, using the principles of Section 7.2.2.

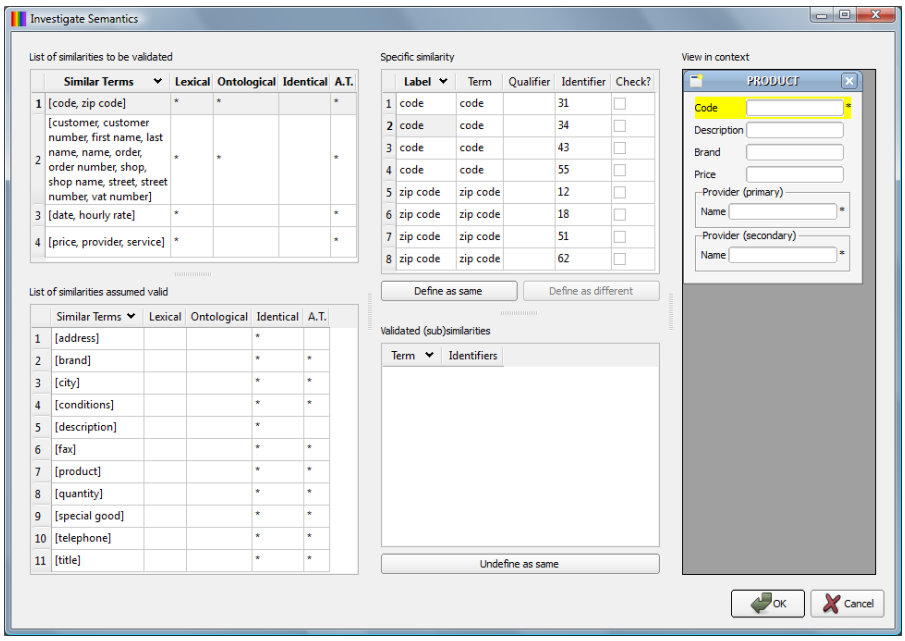


Figure 11.3: Arbitrating terminological similarities.

The sets that are discovered are compared with the previously arbitrated similar subsets, which are stored in the DB-Main repository using meta properties, so that valid pre-existing decisions are maintained. The sets are subsequently presented for arbitration, as illustrated in Fig. 11.4.

For each entity type, the set of structurally similar entity types are presented through their associated form-based interfaces. The structurally similar containers and the shared patterns are highlighted, thanks to the mapping between the data elements and the interface widgets. For entity types that are equal or that unites, a unifying term can be provided.

Based on the provided decisions, the schema is subsequently pre-integrated to reflect them, and the new decisions are stored in the DB-Main repository using meta properties. The terminology of the forms is updated accordingly.

11.4 Providing data samples and constraints

The structurally updated schema is then copied into a new schema to perform the constraint analysis. The previously provided data samples and constraints are loaded from the DB-Main repository, and the valid constraints and depen-

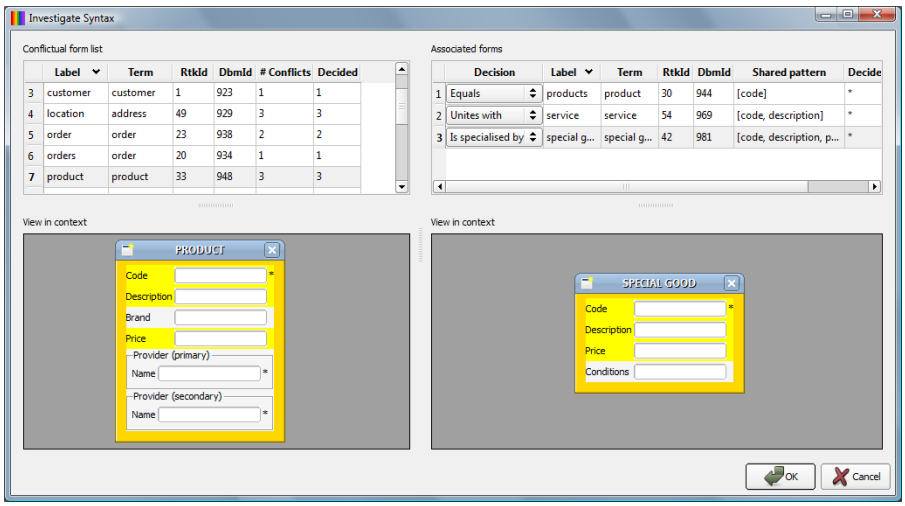


Figure 11.4: Arbitrating structural similarities.

dencies are initialised accordingly, using the principles exposed in Section 8.4.2.

The toolkit enables to handle the entity types associated with one form at a time, as illustrated in Fig. 11.5. For each entity type of each form, the end-users may provide several data samples, that must respect the previously enforced constraints and that will restrict the valid constraints that can be enforced, which notably provides an interactive means to approximate Armstrong relations. At any time, the user can switch from the list of valid data samples to one of the constraints panel.

For the technical constraints, he can visualise the enforced and valid cardinality and prerequisite components for every components of a given entity type, as well as value types for attributes, as illustrated in Fig. 11.6. Any valid constraint can be enforced, and any modified constraint can be reinitialised as long as the set of data samples is compatible. On the contrary, the value sizes are not handled.

Similarly, for the existence constraints, he can visualise the enforced and valid constraints for a given entity type as illustrated in Fig. 11.7. Any valid constraint can be enforced, while any enforced constraint can be unenforced.

The functional dependencies panel contains the enforced, valid and discarded dependencies for a given entity type, as illustrated in Fig. 11.8. The valid dependencies can be enforced or manually discarded, and the enforced dependencies can be unenforced. Discarded dependencies can be reinitialised to recalculate the dependencies that are still valid after all. The dependencies

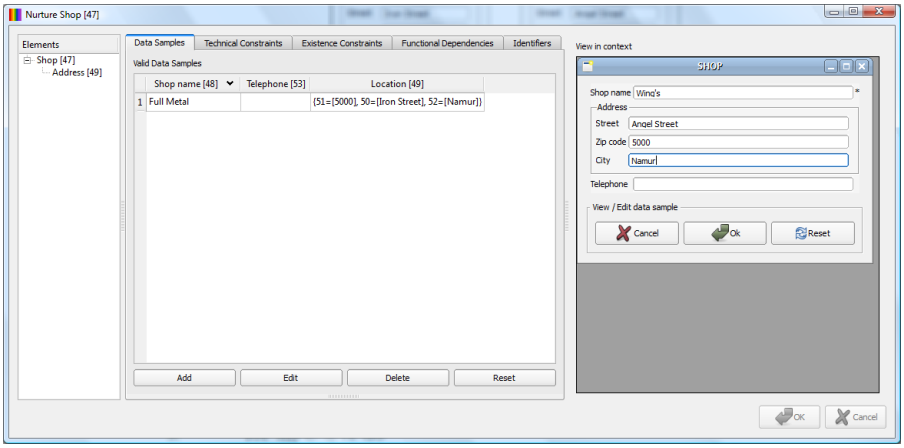


Figure 11.5: Adding data samples.

Valid Constraints

	Shop name [48]	Telephone [53]	Location [49]
Cardinality	Exactly one (1-1)	At most one (0-1)	At most one (0-1)
Cardinality could be	/	/	Exactly one (1-1)
Value Type is	text	text	/
Value Type could be	/	boolean date datetime integer real	/
Requires	/	/	/
Could require	/	/	/

Figure 11.6: Arbitrating technical constraints.

are filtered according to the principles exposed in Section 8.4.4, and set visible (they are formatted using strike-out in the figure) or hidden.

Finally, the identifiers panel contains the enforced (primary and secondary) and valid unique constraints for a given entity type, as illustrated in Fig. 11.8. The constraints can be enforced and unenforced, and set to primary or secondary.

Based on the provided decisions, the pre-integrated schema is updated with the new constraints and dependencies, while the new decisions and constraints are stored in the DB-Main repository using meta properties. The constraints of the forms are updated accordingly.

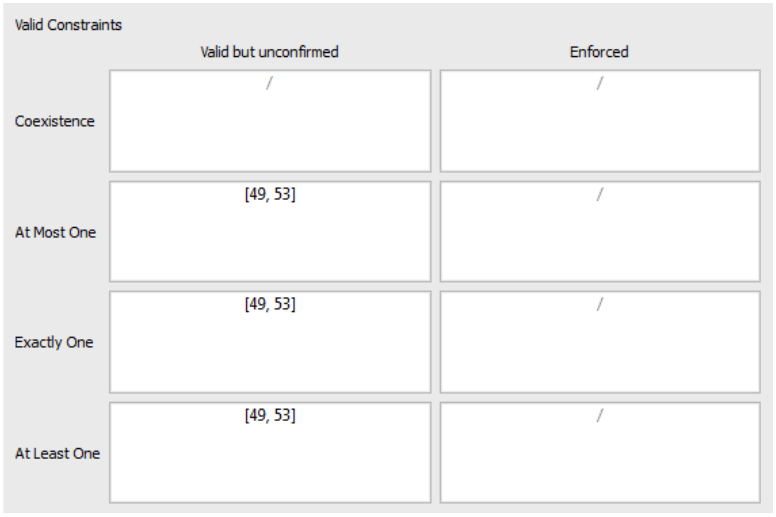


Figure 11.7: Arbitrating existence constraints.

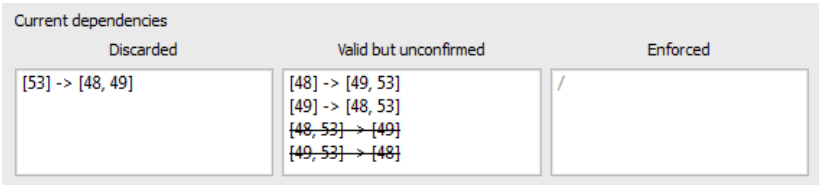


Figure 11.8: Arbitrating functional dependencies.

Figure 11.9: Arbitrating unique constraints.

11.5 Finalising the project

The final process handled by the toolkit is the binding process, which is partially supported. The nurtured pre-integrated schema is copied into a new schema to perform this integration process. Using the principles exposed in Section 9.2.1, the IS-A hierarchies are identified are presented for upward integration and subsequent constraints and dependencies handling, as illustrated in Fig. 11.10.

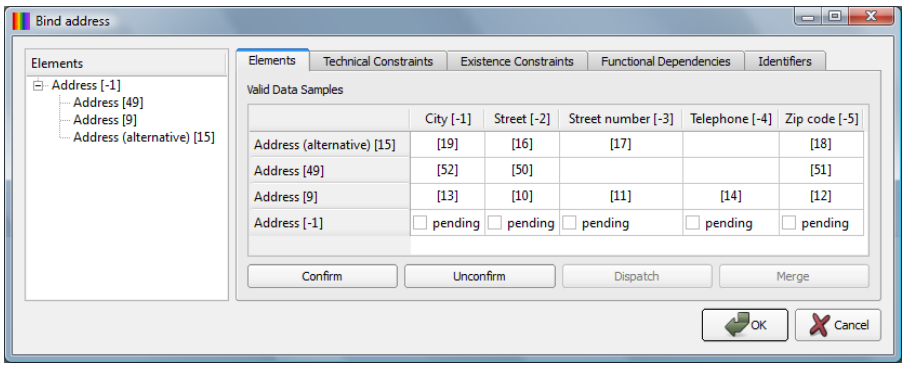


Figure 11.10: Binding concepts.

For each level of the hierarchy, the components that could be inherited from the subtypes are grouped and proposed for integration. Once integrated, the associated constraints and dependencies can also be confirmed or rejected. The schema is subsequently transformed to reflect the provided decisions and represent an integrated conceptual schema of the application domain. The terminology of the forms is updated accordingly.

In contrast, the handling of referential components and dispatching of attributes into relationship types must be handled manually in the DB-Main toolkit, while ensuring the mapping between the widgets and the integrated elements of the data model. However, the issue of referential components can be circumvented upstream. If an entity type is found to complement another during the analysis of structural ambiguities, the users can loop back to edit the drawing step to change the structure of the form and add a container to hold the appropriate referential components, as explained in Section 7.2.4.

Part III

Validation

In this part of the discussion, we address the validation of the RAINBOW approach. Chapter 12 presents the two main research questions underlying the RAINBOW approach, which are (1) its ability to help end-users and analysts to communicate static data requirements to each other, and (2) the quality of the conceptual schemas produced using it. Both questions raise number of issues, and such problems are not easy to experiment, measure and validate, especially given the immanent difficulty of evaluating methodologies for the development of large systems. Since an extensive experimentation is not feasible at our level, an experimentation and validation canvas is define instead and applied to two preliminary case studies, in order to validate the said canvas and coincidentally get a first insight on the implementation of the RAINBOW approach. Chapter 13 subsequently describes the experimentation process, and Chapter 14 discusses its results.

Chapter 12

Validation protocol

In this chapter, we present the two main research questions underlying the RAINBOW approach, and define an experimentation and validation protocol to address them. For this purpose, we briefly review different types of data collection techniques, then formalise the goals and context of the expected experimentation before defining and detailing the protocol itself.

12.1 Research questions

One of the most critical aspect of this doctoral research concerns its validation. The transversal nature of our approach, as well as the interdependence between the methodology and the tool support, naturally lead to two critical research questions. First of all, *does the RAINBOW approach and tool support help end-users and analysts to communicate static data requirements to each other*? Or in other words, do they give the means to express, capture, discuss and validate conceptual schemas, knowing that the stakeholders may have such different backgrounds? This question inherently raises methodological, practical and sociological issues: were our design decisions strategic, or should we have used alternative techniques? Does our approach carry a real added value for stakeholders? Is the current tool support usable and efficient? How does our approach influences the communication between the end-users and the analysts and their behaviour during the whole process?

The second research question concerns the *quality of the conceptual schemas produced using the RAINBOW approach*. How can we define the quality of a schema in general, and the acceptability of these schemas in particular? What

are their flaws and merits, and what could and should be done to improve them? This issue addresses the predicament of quality assessment, which is itself intrinsically complex as well.

Indeed, such problems are not easy to experiment, measure and validate, especially given the immanent difficulty of evaluating methodologies for the development of large systems. To obtain relevant results and draw valuable conclusions, we would need to compare the RAINBOW approach to existing ones, based on multiple experimentations led on numerous and different case studies over an extensive time span. Such an effort is not feasible at our level, and would easily make for a significant research subject *per se*.

However, one of the contributions of this research is instead to define an experimentation canvas, based on preliminary studies that could in turn lead to a more realistic experimentation endeavour. The objective of the experimentation is therefore to *analyse* the use of the RAINBOW Toolkit *for the purpose of* evaluating and improving the RAINBOW approach, *with respect to* its usability and effectiveness as a two-way communication channel *from the viewpoint of* potential end-users and analysts *in the context of* static data requirements elicitation within a process of software engineering.

12.2 Types of data collection techniques

The first concern regarding the validation of the RAINBOW approach was to choose appropriate techniques to lead the preliminary field studies. As presented by [Singer et al., 2008], there are various types of data collection techniques for software engineering, which can be classified into three categories, namely direct, indirect and independent.

Direct techniques make the researchers interact more or less directly with the participants. Among these techniques, *inquisitive techniques* (such as brainstorming, focus groups, interviews, questionnaires, ...) are useful to obtain a better understanding of processes by getting to debate with their very actors. This can also lead to discuss the tools and conditions in which the processes are performed and how they are experienced by these actors. *Observational techniques* (such as shadowing and participant-observer) rather focus on monitoring the studied phenomena to describe and analyse them as accurately as possible. Both categories of techniques are quite time-consuming for all persons involved, and their output must be handled with caution, since they can be subjective and modified by the previously mentioned Hawthorne effect [Mayo, 1933].

Indirect techniques focuses on the working environment and equipment of processes, which implies that the researchers do not interact with any partici-

pants. The actors of the processes continue their work while data is collected either transparently (through instrumentation systems) or voluntarily ("fly on the wall" technique, where the participants record their own work), so that it can be analysed afterwards. These techniques are appropriate for longitudinal studies and require little to no additional time from the participants.

Finally, *independent techniques* are suitable for processes that can be seen as black boxes from which the products can be analysed. With these techniques, the researchers only access work artefacts related to the participants (such as change logs, tool logs, bug trackers, documentation, code, execution traces, ...), in order to uncover information on how these participants get their work done.

Selecting the appropriate techniques depends on the research questions, as well as the available participants and artefacts for the study. Let us therefore review the goals and the context of our experimentation.

12.3 Goals and context of the experimentation

As explained, the main validation concerns relate to the ability of the RAINBOW approach to enable the communication of static data requirements between stakeholders of a software engineering project, and the quality of the requirements produced using our approach. The experiment should therefore focus on two primary goals that can be refined as follows.

12.3.1 Goal 1: Assessing the effectiveness of the RAINBOW approach

The first goal concerns the effectiveness of the RAINBOW approach as a means to transparently communicate static data requirements between end-users and analysts. In other words, we want to:

- *Analyse* the use of the RAINBOW Toolkit as tool support of the RAINBOW approach
- *For the purpose of* assessing their effectiveness to express, capture and validate static data requirements
- *With respect to* their usability and potency to generate discussions
- *From the viewpoint of* end-users and analysts
- *In the context of* static data requirements elicitation within a process of software engineering.

Basically, we will consider the approach effective if the end-users consider that they were easily able to express all their static data requirements using it, and if the analysts consider that the approach helped them to get a good

understanding of the application domain and the subsequent database that will need to be developed. The analysts appreciation of the elements generated through the approach rather regards the quality of the approach's output, which we will discuss afterwards. More precisely, we need to be attentive to the following *efficiency-related elements*:

- the possible *articulation* problems that were presented in Section 2.1, namely: confusion, improper expectations, difficult or unclear articulation, inappropriate prioritisation;
- the attitude and satisfaction of the participants regarding the methodology (and how they can possibly compare them to other approaches);
- the information that could and could not be expressed using the toolkit;
- the discussions that were induced by the approach and toolkit for the requirements that could and could not be expressed using the toolkit;
- the ease of use and reliability of the toolkit;
- the relevance of the elements presented by the toolkit (similar labels and structures, possible constraints and dependencies) for end-users arbitration.

12.3.2 Goal 2: Assessing the quality of the RAINBOW output

The second goal concerns the quality of the schemas produced using the RAINBOW, to understand if these schemas are relevant, usable and useful. In other words, we want to:

- *Analyse* the output of the RAINBOW Toolkit as tool support of the RAINBOW approach
- *For the purpose of* assessing their effectiveness to produce static data requirements
- *With respect to* their quality
- *From the viewpoint of* analysts
- *In the context of* static data requirements elicitation within a process of software engineering.

Defining the quality in Software Engineering in general, and in Data modelling in particular, is an old, complex and never-ending issue, though standards such as the ISO/IEC 9126 have been established [ISO/IEC, 2001]. The notion of quality can focus on various elements, such as the form and the content of various artefacts (data models, code, applicative components, ...) or specific characteristics of these artefacts (such as maintainability, evolvability, performance, ...). According to the peculiar aspects that need to be studied, four

main trends have appeared to address this topic. Let us present very briefly present these approaches and provide a few references for further investigation:

- *Frameworks* allow their authors to tackle quality of models based on theoretical methods (see for instance [Briand et al., 1996], [Hoxmeier, 1998], [Lindland et al., 1994], [Krogstie, 1998], [Kesh, 1995], [Habra et al., 2008], [Maes and Poels, 2006] or [Moody et al., 2003; Moody and Shanks, 2003]). These frameworks can typically be (non exclusively) classified as:
  - semiotic, which addresses the quality of models and/or modelling languages based on how their syntactics (syntax), semantics (expected meaning) and pragmatics (actual interpretation) combines with different elements of context of use;
  - methodological, which relies on practical methodologies and processes for evaluating quality;
  - relational, which focuses on the interactions between quality factors;
  - model-based, where the quality is analysed through structured dimensions that can themselves be structured through attributes and properties;
- *Metrics* enable quality to be evaluated based on mathematical functions, typically involving different object counts (number of associations, number of entity-types, ...) constrained by specific coefficients (see for instance [Genero et al., 2000] or [Cherfi et al., 2002]);
- *Best practises* are built on empirical evidence to usually suggest visual and structural improvement for models (see for instance [Moody, 2006], [Gemino and Wand, 2005] or [Daly et al., 1996]);
- *Analytical approaches* focus on specific quality problems such as normalisation, minimality or structural consistency, and resolve them through theoretical reasoning (see for instance [Codd, 1971b,a] or [Wahler, 2008]);

Obviously, systematically reviewing all the approaches in that domain and defining an extensive canvas to evaluate the quality of the schemas produced using the RAINBOW would be a tremendous endeavour that could make for a whole research topic in itself. However, if we recall the expected characteristics of a *Software Requirement Specification* (as detailed in Section 2.1), we can nevertheless observe that they coincide with most of the criteria often mentioned in quality-related researches.

Simply speaking, we will therefore consider that the output of the RAINBOW approach is of quality if the analysts consider that they gathered all the static data requirements necessary to build an appropriate and reliable

database. We hence need to be attentive to the following *quality-related criteria*:

- *correctness*: does the schema use appropriate constructs?
- *consistency*: is the schema free of contradictions?
- *completeness*: does the schema cover (exactly) all the aspects necessary to conceive the future database of the software engineering project (*scope*), and is it detailed enough (*level of details*)?
- *conciseness*: is the schema free of redundancies?
- *unambiguity*: are there elements of the schema that are still unclear or disputable?
- *modifiability*: can the schema be updated easily ?
- *traceability*: can each element of the schema be retraced to the original requirements expressed by the end-users?
- *verifiability*: can the schema be used to verify that the software meets the requirements?
- *testability*: can pass/fail or quantitative assessment criteria can be derived from the schema?

Besides, we also want to analyse the following *practical issues*:

- Does the approach help the analyst to understand the application domain, whether he was part of the experimentation or not?
- What could and should be done to improve the output schemas?

12.3.3 Context of the experimentation

To answer these questions, the use of the RAINBOW approach and toolkit during the experimentation should be as faithful as possible to their expected context of use in the real life and in a wider experimentation context, which implies that we ought to be careful to the following aspects, some of which are mentioned in Section 4.2:

- *Application type*: the modelled software engineering projects should be related to form-based data intensive applications;
- *Type of company*: the projects should involve few people to reflect that the expected companies are small to medium sized;
- *Experience of the participants*: the end-users should be familiar with form-based human-computer interactions and the analysts should be familiar with (static) data modelling;

- *Analysed Process*: the experimentation should focus on Data Requirements Engineering within any Software Engineering process (Waterfall model, Iterative and Incremental Development, Agile Development, ...);
- *Tools*: we use the RAINBOW Toolkit.

To understand more precisely the context of the experimentation, let us also recall that this research focuses on Database conceptual modelling [Batini et al., 1992; Hainaut, 2006] (as part of Requirements Engineering), in conjunction with Database reverse engineering [Chikofsky and Cross, 1990; Hall, 1992], Prototyping [Gomaa and Scott, 1981; Lantz, 1986] and Participatory Design [Schuler and Namioka, 1993].

12.4 Building our dedicated validation approach

12.4.1 Overview

As already hinted, evaluating the RAINBOW approach it is not a trivial matter. First of all, evaluating methodologies for the development of large systems requires methodological comparisons on a significant amount of case studies among lengthy periods of time, which we cannot afford. Secondly, comparing methodologies on the same or a limited number of case studies is also problematic. If we apply the RAINBOW approach and another approach on the same case study, there will be inevitably interferences and biases, depending on the sequence of the possible (inter)actions of each approach. Leading two instances of the same case study in parallel is also questionable, since we would have to separate the end-users into two groups which could influence their behaviour as subgroups. As for leading different case studies with different approaches, there would be little left for relevant comparison and analysis. Finally, it is intrinsically difficult to precisely define and objectively measure the *effectiveness* of such a transversal method and the *quality* of the schemas it produces.

Since we wanted to draw guidelines for a wider experimentation and since there is no indisputable experimentation solution, we therefore chose to observe and assess real-life implementations of our approach. To do so, we defined two independent studies S1 and S2, based on real-life issues concerning two end-user participants EU1 and EU2, and decided to use the *Participant-Observer* principles to monitor the use of the RAINBOW toolkit and approach, and the *Brainstorming/Focus group* principles to analyse the resulting conceptual schemas.

Therefore, for each preliminary study, a pair of observers (including a main observer MO and a different assistant observer in each case, namely AO1 and

AO2) observed the interaction of one of the end-users with an analyst DB1 (the same in each case), jointly designing the conceptual schema of their dedicated project using solely the RAINBOW methodology and toolkit. The role of the observers was to follow the process and take note of all the situations where the usage of the methodology and toolkit were efficient or not.

Then, each resulting conceptual schema was discussed by three database analysts (DB1, DB2 and MO) to determine their qualities and flaws, as well as the delta between the "automatically" produced schemas and a likely "improved" version.

12.4.2 Participants

Seven participants were therefore involved, namely two end-users (EU1 and EU2), two analysts specialised in Database Engineering (DB1, DB2), and three analysts playing the role of observers (OB1, OB2 and MO), the latter being also specialised in Database Engineering. All participants are employed at the Faculty of Computer Science of the University of Namur, Belgium, and were chosen because of their profile:

- EU1 is the secretary notably responsible for promoting the teaching programmes of the Faculty of Computer Science and handling the registration of applicant students since 16 years. After obtaining her Bachelor Degree in Computer Science, she previously worked during 25 years as a researcher is process engineering. She is therefore aware and used to human-computer interaction, though most of the current applicant files are still received through postal mail. Her motivation lies is that she would be interested in improving the registration process of the applicant students.
- EU2 is the executive secretary of the Faculty of Computer Science since 7 years, and carries other tasks such as the organisation of seminars, symposiums and other events for the Faculty. Previously, she worked five years as a secretary for a non-profit-making organisation, before becoming the secretary for the Teaching Units of the Faculty of Computer Science for 16 years. She never received any proper computer training, but masters usual office tools such as Word, Excel, PowerPoint or FileMaker, which gives her substantial knowledge in (form-based) human-computer interaction. Her motivation lies is that she would be interested by a tool facilitating the organisation of events for the Faculty.
- DB1 is a researcher and PhD student of the Laboratory of Database Application Engineering (LIBD) of the Faculty of Computer Science since 3

years. He has a Master Degree in Computer Science and his doctoral research focuses on Database Quality, with a peculiar interest in the quality of conceptual schemas.

- DB2 is a researcher and lecturer of the Laboratory of Database Application Engineering (LIBD) of the Faculty of Computer Science since 12 years, after obtaining her Master Degrees in History and Computer Science. During her tenure, she participated in several projects for which she handled the database administration, and was notably a member of the previously mentioned ReQuest project. In addition to her involvement in the latter project, she has a special interest for integration and normalisation issues.
- OB1 is a teaching assistant and PhD student of the Faculty of Computer Science since 5 years, after obtaining his Master Degree in Computer Science. During his tenure, he has been the leading assistant for the course of Software Engineering of the Faculty of Computer Science and is leading a PhD related to the quality of the information flow within the Software Engineering process.
- OB2 is a teaching assistant and PhD student of the Faculty of Computer Science since 2 years. He has a Master Degree in Computer Science and previously worked as a researcher for the Tokyo University of Agriculture and Technology during 3 years. His is specialised in the Computer Networks, but he has a special interest in human-computer interaction and software engineering.
- MO is actually yours truly, and the initiator of this validation process. He is a teaching assistant and PhD student of the Laboratory of Database Application Engineering (LIBD) of the Faculty of Computer Science since 3 years. After obtaining his Master Degree in Computer Science, he was a member of the previously mentioned ReQuest project for four years and is currently the main assistant for the Database courses and one of the assistants for the Software Engineering course of the Faculty of Computer Science.

We deliberately chose to assign DB1 to carry both studies in order to prevent possible biases and variations. On the one hand, he had no prejudice on the RAINBOW approach, on contrary of MO, and on the other hand, he was neutral towards EU1 and EU2, while DB2 is personally closer to EU1 than EU2. In the contrary, we chose two different neutral observers to support the main observer MO in order to possible notice different types of phenomena. Each of these participants were assigned different types of tasks, that we will now detail.

12.4.3 Task 1: Preparing the experimentation

Before beginning the evaluation, a different software engineering project was defined with each end-user. Before starting the study, a separate half-hour meeting was therefore organised with each end-user to define the overview and goals of their project. From that point on, the following rules were established and agreed upon to avoid biases and interferences between the studies, but also to to keep the focus on the sessions and canalise the efforts during those laps of time:

- *Rule 1: Seal of Secret.* The two studies are anonymous and independent, and the participants are not supposed to talk to each other about the experiment if the main observer (MO) is not present, even if they are part of the same study. However, a mail can be sent within the members of a same study to discuss it, as long as MO is put in carbon copy.
- *Rule 2: Only there and then.* The participants are not supposed to use the RAINBOW toolkit outside the sessions, for instance to modify what they produced during the previous session. To prevent this to occur, the application will only be available on the laptop of MO.
- *Rule 3: The observers do not exist.* As far as possible, the end-user and the analyst must act as if the main observer and his assistant observer are not present.

Before starting the observation, EU1, EU2, DB1, OB1 and OB2 received a short training on the tool support and methodology based on 2 screencast tutorials explaining how to draw form-based interfaces using the RAINBOW toolkit (see Appendix C). DB2 was not involved in this process in order to preserve her neutrality for the last task.

12.4.4 Task 2: Applying the RAINBOW approach to each project

For each study, the pair of end-user and analyst were asked to jointly design the conceptual schema of their application project using the RAINBOW methodology and toolkit, while the observers took notes. This process was organised in four sequential steps:

1. *Drawing the forms (REPRESENT)*: first of all, the end-users had to draw and edit forms that would allow them to accomplish usual tasks of their future application project. They were asked to focus on the terminology and data aspect of this application rather than the layout and general appearance of the forms. In particular, they had to pay special attention to the consistency of the labels/terms and the specification of the widgets

they needed, typically to input data. During this process, they were expected to control the RAINBOW toolkit, while the analyst assisted and guided them whenever necessary.

2. *Analysing the terminology and structure of the forms (INVESTIGATE)*: (1) the end-user and the analyst first had to analyse the terminology of all the form elements to clarify any remaining ambiguities; (2) then, they had to analyse the terminology of the containers to explain the relations existing between these containers. If necessary, the pair could go back and edit their forms. During this process, the end-user could still operate the RAINBOW toolkit or give the control to the analyst. Whenever necessary, the pair could go back and edit their forms.
3. *Provide examples and constraints (NURTURE)*: for each form, the pair first had to provide a set of examples, then examine the technical constraints, the existence constraints, the functional dependencies and the possible identifiers associated with the form and its elements. If necessary, the pair could go back and replay the previous steps.
4. *Finalize the project (BIND)*: from the previous steps, a set of "high level" concepts were materialised. For each of these concepts, the pair had to arbitrate the properties that were to be associated with the concept, then examine the associated technical constraints, the existence constraints, the functional dependencies and the possible identifiers. If necessary, the pair could go back and replay the previous steps.

For each study, one session of two hours per step per week was planned. Each session was organised as follows:

- *Introduction*: the main observer (MO) recalled the previous steps and presented the main objectives of the current session;
- *Recapitulation*: the participants discussed the previous steps and the possible elements that remained unclear or that should be reworked;
- *Execution*: the end-user and the analyst executed the tasks associated with the current session using the RAINBOW toolkit while the observers took notes;
- *Individual debriefing*: afterwards, the main observer discussed separately with each other participant to take their impressions and remarks.

Recall that the ADAPT step is automatic, and that we deliberately left aside the OBJECTIFY and WANDER steps, since the generation of the components is straightforward and the manipulation of a reactive prototype mainly adds another level of validation. Throughout these steps, DB1 never saw the

conceptual schemas that were produced and DB2 was not involved in this process in order to preserve her neutrality for the last task.

During those sessions, the observers were asked to be attentive to the efficiency-related elements that were introduced in Section 12.3.1, as well as any other element that they felt relevant to the efficiency of the approach and tool-support.

12.4.5 Task 3: Debating the quality of the produced schemas

At this point, the RAINBOW toolkit had produced a *pre-integrated* schema for each study, while DB1 drew the conceptual schemas he felt the most appropriate for each study before without at the output schemas. An additional step was subsequently held between DB1, DB2 and MO to discuss the qualities and the flaws of these schemas obtained using the RAINBOW approach.

During those sessions, the analysts were asked to be attentive to the quality-related criteria and issues that were introduced in Section 12.3.2, as well as any other element that they felt relevant to the quality of the schemas produced using the approach and tool-support.

Chapter 13

The experimentation

In this chapter, we present the execution of the two preliminary studies that were led. The first one concerns the electronic support of students wanting to apply for studies at the Faculty of Computer Science of the Namur University, while the second one deals with the definition of an academic event management system. After preparing the experiment by defining the subject and training the participants, we present how each pair of end-user and analyst managed to jointly design the conceptual schema of their application project using the RAINBOW methodology and toolkit, while observers took notes about the efficiency of the process. We then expose the subsequent discussions on the quality of the schemas produced using the approach and tool support for each study. Both case study were led in French, but for the sake of this dissertation, we translated the resulting forms and schemas. See Appendix C for the original materials.

13.1 First case study: A student application form

13.1.1 Preparation

Defining the subject

During the preparation of the experiment, EU1 explained that, at the time being, students (and in particular foreign students) who would like to apply for studies at the Faculty of Computer Science of the Namur University needed to fill then fax a paper form with various personal information and details about their educational and professional curriculum. Moving this form to an

electronic medium would enable for instance to handle a web application for the registration of such demands. We therefore agreed to carry this case study in order to define the conceptual schema supporting such a project.

Training

Before the beginning of the sessions, EU1 received the two screencast tutorials explaining how to use the RAINBOW toolkit to draw relevant form-based interfaces. She did not have many remarks, except that she found the notion of "parent-dependent" unclear and possibly conflicting with the one of cardinality. She also suggested the tutorial to be provided with a locale translation and more navigational buttons (for instance to go back and skip sections).

13.1.2 Session 1: Drawing the forms

The first session focused on drawing the necessary forms to encode all the data relative to a new application for a computer science student applicant, and was organised as follows:

- *Introduction*: the main observer (MO) introduced the participants and their roles, as well as the objectives (drawing the forms supporting the encoding of information for the subject defined by EU1) and the procedure (EU1 and DB1 draw jointly while OB1 and MO observe) for the current session (10 minutes);
- *Recapitulation*: the participants recapped and discussed the subject of the study, and discussed the tutorials (10 minutes);
- *Execution*: the end-user and the analyst executed the tasks associated with the current session using the RAINBOW toolkit while the observers took notes (90 minutes); Note that a paper form existed before our validation process, and the end-user had beforehand sketched a paper form to collect her ideas regarding what should appear in the drawings;
- *Individual debriefing*: afterwards, the main observer discussed separately with each other participant to take their impressions and remarks (15 minutes per participant).

In the following, we expose the remarks made by the observers and the participants through the debriefing.

General observations

The execution of the drawing took 90 minutes altogether. However, the end-user and the analyst started showed sign of fatigue after 65 minutes, which

suggests that the duration of the session was too long.

The RAINBOW toolkit was installed on MO's laptop, with which EU1 and DB1 were not accustomed. EU1 acknowledge that she would have preferred working on her own desktop computer.

When the drawing started, the end-user naturally gave the commands to the analyst and was initially reluctant to manipulate the rainbow toolkit, but she eventually took up the reins of the prototype. On the other hand, the analyst did not find this configuration gratifying, because he did not feel very useful or required for the process.

Still, the end-user acknowledged that the presence of the analyst was helpful, because of the advices and explanations that he regularly provided. Rather than being a user-driven design session, the drawing of the interfaces turned to be a joint development, and using the tool led to discussions on the form and substance of the interfaces.

Observations on the tool support

It appeared that the automatic suggestions of the integrated label analyser actually annoyed the end-user, who found them too intrusive and most of the times irrelevant. She found that her flow was interrupted, and would have preferred clarifying the labels subsequently.

The automatic graphical rendering of the widgets according to its properties surprised the end-user, especially the selection widget. The analyst had to insist that one of the motivations behind this tool behaviour was to lead the end-user to focus on the content of the forms rather than their appearance.

The available widgets are restricted to forms, fieldsets, tables, inputs, selections and buttons, and for the given problem, these widgets seemed sufficient, though we can observe that no button was used. We also observe that the though the problem could have been reduced to smaller sub problems (typically encode a student, then encode an application for this student), the end-user drew a single form to collect all the data.

We observed that the end-user drew 3 different tables for the same type of issue (`Academic Year`), instead of merging them into a single table while providing explanations in the "description" property of the table. This could indicate that different possible uses and combinations of widgets may need to be more explicitly explicated.

As previously mentioned, widgets have a cardinality specifying how many values could and should at least and at most be provided. We observed that the end-user often specified widgets as "mandatory", even if she acknowledged that "it would not be that problematic if this field was not filled".

Finally, there was few manipulation mistakes observed during this session. They were mainly relation to the insertion process (widgets inserted after rather and before, and vice-versa). The end-user drew only 3 of the 5 previous `Academic Years`, because she found it repetitive.

Output of the session

During this first session, EU1 and DB1 produced the single form at the left side of Fig. 13.1. This form was automatically adapted into the schema at the right side of Fig. 13.1, before being transformed into the schema of Fig. 13.2 using the previously defined mapping rules.

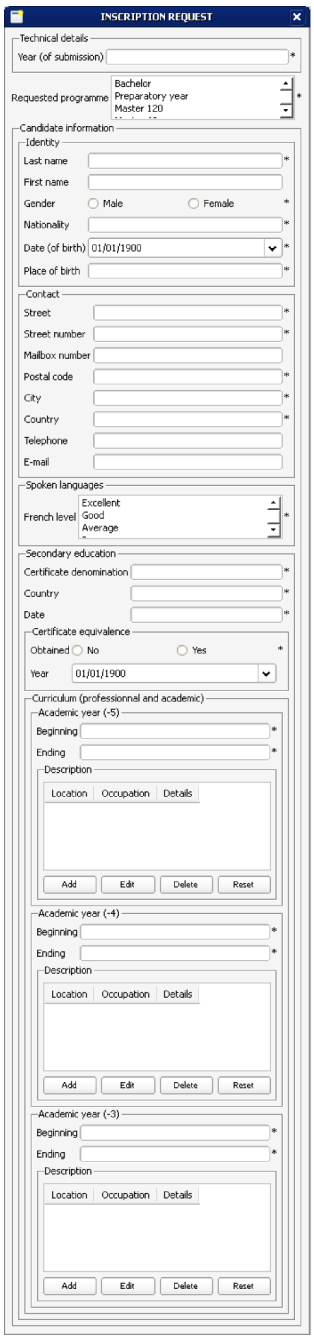
As we can see, this single form represents the inscription request made by a candidate. Each request has a year of submission and a requested programme, and various information must be provided regarding the applicant, notably regarding his/her identity, his/her means of contact and his/her education and training. We observe that this form is quite big, which makes it less pleasant to manipulate.

EU1 enjoyed drawing the form and seeing the interface developing little by little. Though DB1 did not feel very useful, the pair actually discussed throughout the whole session to appropriately define each component of the form.

13.1.3 Session 2: Analysing the terminology and structure of the forms

This session focused on analysing the terminology and structure of the form, in order to detect any possible ambiguity. The session was organised as follows:

- Recap of the objectives of this session
- Discussion on the previously drawn form, with suggestions of modification.
- Analysis of all the labels: the similar labels have been group in lexically or ontologically similar label clusters; in these cluster, any element that represent the same king of information must be grouped and jointly relabelled (30 minutes).
- Analysis of the similar structures: the forms, tables and fieldsets containing widgets with the same labels are presented for comparison and arbitration; the end-user must explain why such situation occurs (equality, specialisation, union, complementarity or accident).



INSCRIPTION REQUEST
- Technical Details[0-1]
  - Year (of submission)
- Requested programme
- Candidate information[0-1]
  - Identity[0-1]
    - Last name
    - First name[0-1]
    - Gender
    - Nationality
    - Date (of birth)
    - Place of birth
  - Contact[0-1]
    - Street
    - Street number
    - Mailbox number[0-1]
    - Postal code
    - City
    - Country
    - Telephone[0-1]
    - E-mail[0-1]
  - Spoken languages[0-1]
    - French level
  - Secondary Education[0-1]
    - Certificate denomination
    - Country
    - Date
    - Certificate equivalence[0-1]
      - Obtained
      - Year[0-1]
    - Curriculum (professional and academic)[0-1]
      - Academic Year (-5)[0-1]
        - Beginning
        - Ending
        - Description[0-N]
          - Location
          - Occupation[0-1]
          - Details
      - Academic Year (-4)[0-1]
        - Beginning
        - Ending
        - Description[0-N]
          - Location
          - Occupation[0-1]
          - Details
      - Academic Year (-3)[0-1]
        - Beginning
        - Ending
        - Description[0-N]
          - Location
          - Occupation[0-1]
          - Details

Figure 13.1: The form drawn by end-user EU1 and analyst DB1 during the first session, and its corresponding raw schema.

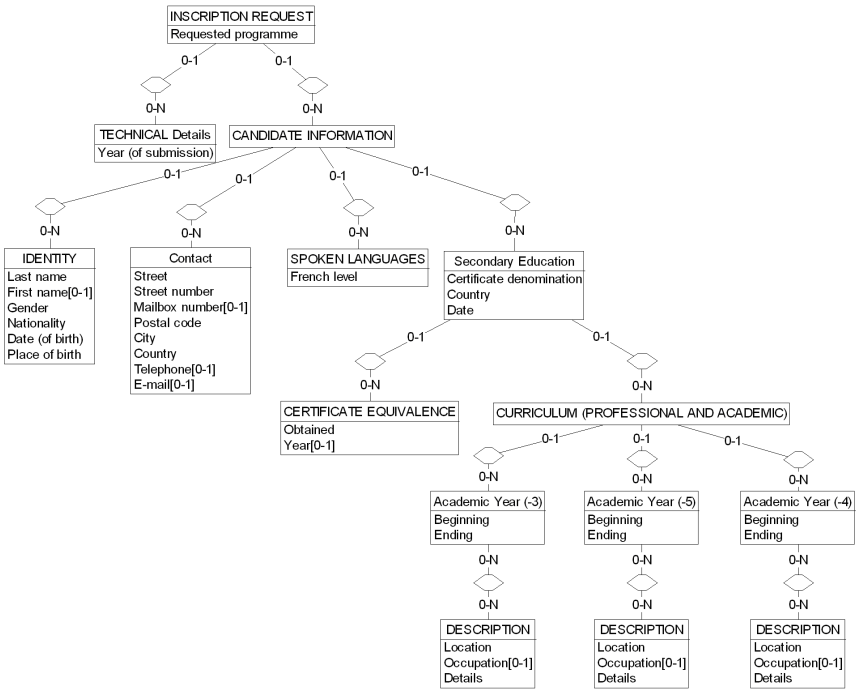


Figure 13.2: The refined schema corresponding to the raw schema of Fig. 13.1.

- *Individual debriefing*: afterwards, the main observer discussed separately with each other participant to take their impressions and remarks (15 minutes per participant).

In the following, we expose the remarks made by the observers and the participants through the debriefing.

Reviewing the interfaces

Based on the notes and discussion from the first session, the analyst suggested modifying the previously drawn form as follows (Fig. 13.3 and Fig. 13.4):

- Synthesize the different `Academic Years` of the `Curriculum` into a single table labelled `Experience` to make the form lighter and more generic. The description property explained that this table refers to the experience of the candidate during the five years prior to the application.
- Remove the `Experience` from the `Secondary Education`, since they were inadvertently mixed up.

These suggestions were agreed by the end-user. In addition to these modifications, the end-user also suggested to relabel `Experience` to `Curriculum`, as show further on. Subsequently, the updated form was "adapted" using the mapping rules, then EU1 and DB1 proceeded with the execution of their task.

Observations on the terminological analysis

The original labelling ambiguities that were detected are presented in Table 13.1. After reviewing these ambiguities, it appeared that there were no labels that still needed to be clarified thanks to the labelling suggestions and the discussions that occurred during the drawing phase. The conflicting elements were therefore different and did not need to be relabelled.

Ambiguities	**Similar sub groups**
Coordonnees-candidat, Donnees-candidat, Donnees-techniques, Identité- candidat	/
Date, detail, e-mail, equivalence de diplome, intituled-du-diplome	/
Lieu	/
Nom, prénom	/
Numero-boite, numero-rue, numero-telephone, rue	/

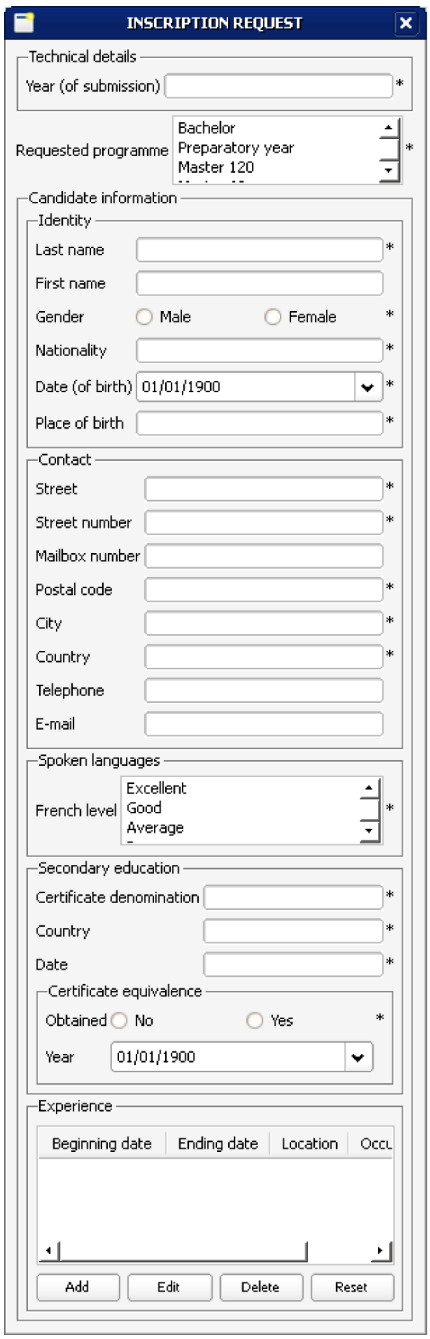
Table 13.1: Labelling ambiguities for session 2 of the first case study.

Observations on the structural analysis

The structural ambiguities that were detected are presented in Table 13.2. After reviewing these ambiguities, it also appeared that the structural ambiguities were purely accidental. Though they shared similarly labelled widgets, the involved containers were actually different.

Ambiguities			**Pattern**	**Decision**
Contact	-	Secondary education	Country	Different
Technical details	-	Certificate equivalence	Year	Different
Secondary education	-	Identity	Date	Different

Table 13.2: Structural ambiguities for session 2 of the first case study.



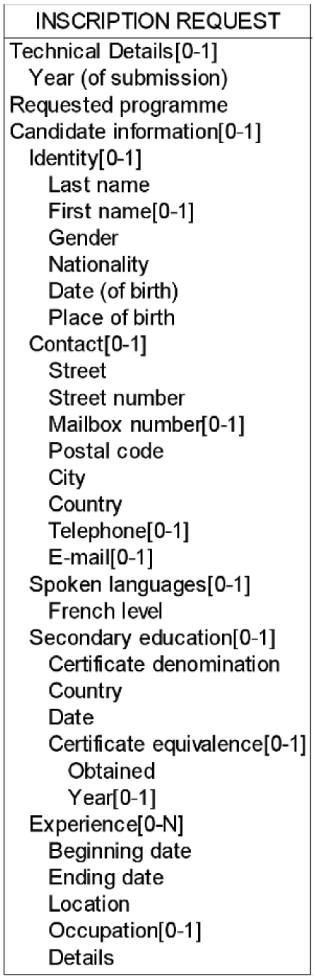


Figure 13.3: The modified form as suggested by the analyst, and its corresponding raw schema.

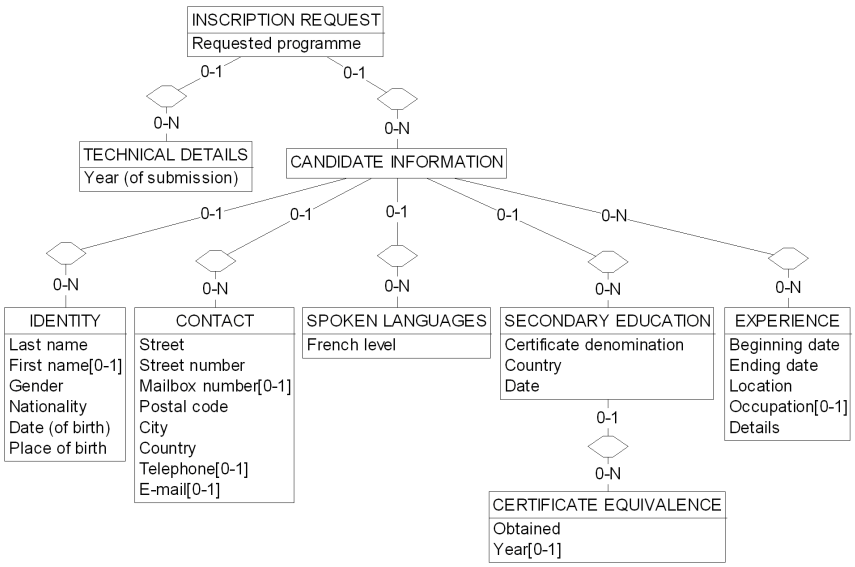


Figure 13.4: The refined schema corresponding to the modified form suggested by the analyst (Fig. 13.3).

Output of the session

The form resulting of this session can be seen in Fig. 13.5, while its underlying schema can be seen in Fig. 13.6.

This session did not provide any modification on the core of the project. However, it led to discuss the general structure and labelling of the form again.

13.1.4 Session 3: Providing examples and constraints

This session focused on providing and analysing examples to discover explicit and implicit properties of the form. The session was organised as follows:

- Recap of the objectives of this session
- Discussion on the previously drawn form, to see if other modifications should be brought.
- Example input and discussion on the properties of the form.
- *Individual debriefing*: afterwards, the main observer discussed separately with each other participant to take their impressions and remarks (15 minutes per participant).

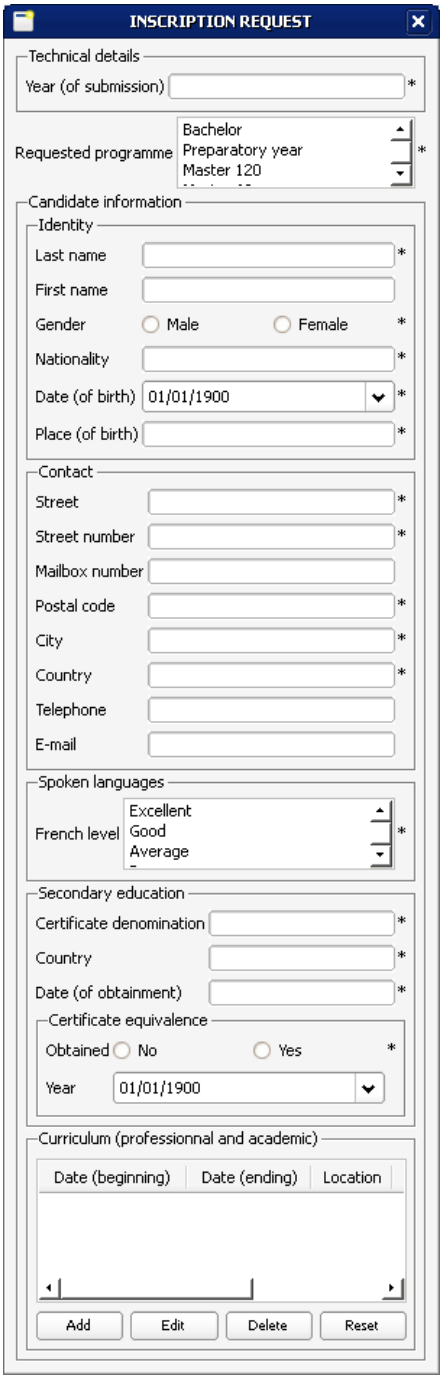


Figure 13.5: The form at the end of the second session.

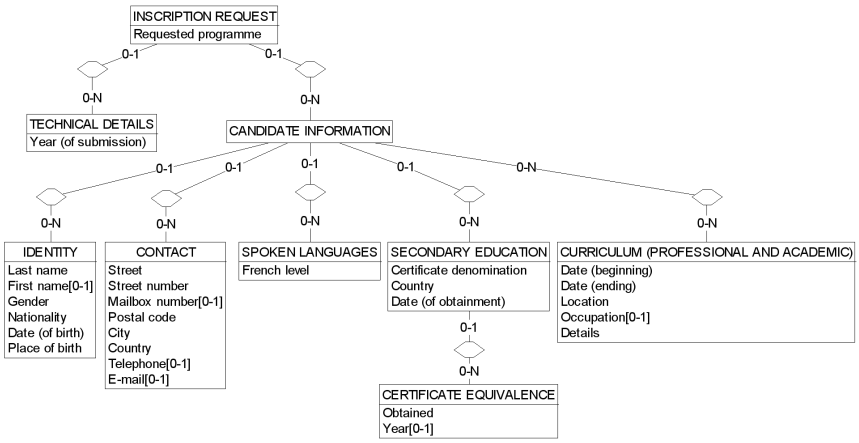


Figure 13.6: The underlying schema of the form at the end of the second session.

In the following, we expose the remarks made by the observers and the participants through the debriefing.

Reviewing the interfaces

At this point of the process, EU1 and DB1 did not feel that the form needed to be updated.

General observations

EU1 took the control of the application to input the examples, based on real paper applications, and during the experimentation, DBA1 eventually took over. EU1 provided 3 examples in more or less 75 minutes, including:

- 15 minutes for example 1;
- 10 minutes for example 2 (which was lost because of a bad manipulation, then recreated by DBA1 in 5 minutes);
- 10 minutes for example 3.

It actually appeared that the encoding of examples was more efficient (quicker and more gratifying) when DBA1 was encoding the data samples under the supervision of EU1, which was reading him the necessary paper forms and documents. The examples were encoded before analysing the constraints and dependencies.

The only interesting constraint that was added was setting `French level` as the identifier of the `Spoken Languages`. Besides, complex constraints could not easily be expressed through the tool:

- complex identifiers: all the `Inscription request` should have be identified by the `Year (of submission)` and the `Identity` of the candidate;
- transversal dependencies: the `Identity` of the candidate determines his `Secondary education`;
- multiple coexisting elements: there should have been at least be a contact means (an address, a telephone number or an e-mail) for each `Inscription request`;
- conditional elements: another dependency that could not be expressed concerned the fact that `Certificate equivalence` should be completed only if the `Country` of `Secondary education` is not European.

Since working with the tool raised these issues, the analyst was at least able to note them for further notice, which implies that those information were not lost. Further discussion between EU1 and DB1 confirmed that there should not be any other "hidden" constraint among the given elements.

Output of the session

At this point, the appearance of the form has not changed, but the underlying schema has (Fig. 13.7). It now includes the specified identifier and has been annotated with the known domain of values and additional information provided by the end-user.

During this session, EU1 hence provided a set of examples in order to elicit constraints and dependencies. However, the application domain was such that it was difficult to point out any major constraint or dependency.

13.1.5 Session 4: Finalising the project

This last session should have focused on discussing the main concepts emerging from the forms, especially the one highlighted during the structural analysis. However, no major concept was detected, and therefore this last session became obsolete.

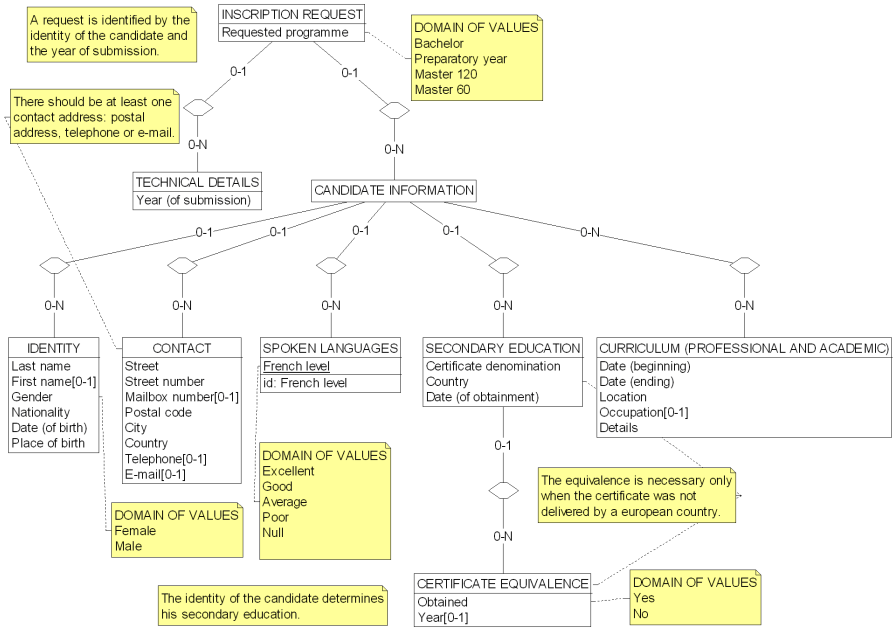


Figure 13.7: The underlying schema of the form at the end of the third session.

13.1.6 Discussing the schemas

Characteristics of the subject

This case study was highly interesting because it dealt with a simple case study that could be represented by a single monolithic form. There was very little redundancies and ambiguities, yet there were some interesting constraints to be expressed.

The subjected reviewed by the analyst

Based on the form-based interfaces and the knowledge he gathered during the different steps of the approach, the analyst DB1 drew his own schema of the application domain before analysing the schema generated by the RAINBOW toolkit. The result of his modelling can be seen in Fig. 13.8.

Analysing the generated schemas

As we can see, the final generated schema is basically a tree of entity types, reflecting the tree-like structure of the form. The content is fundamentally the

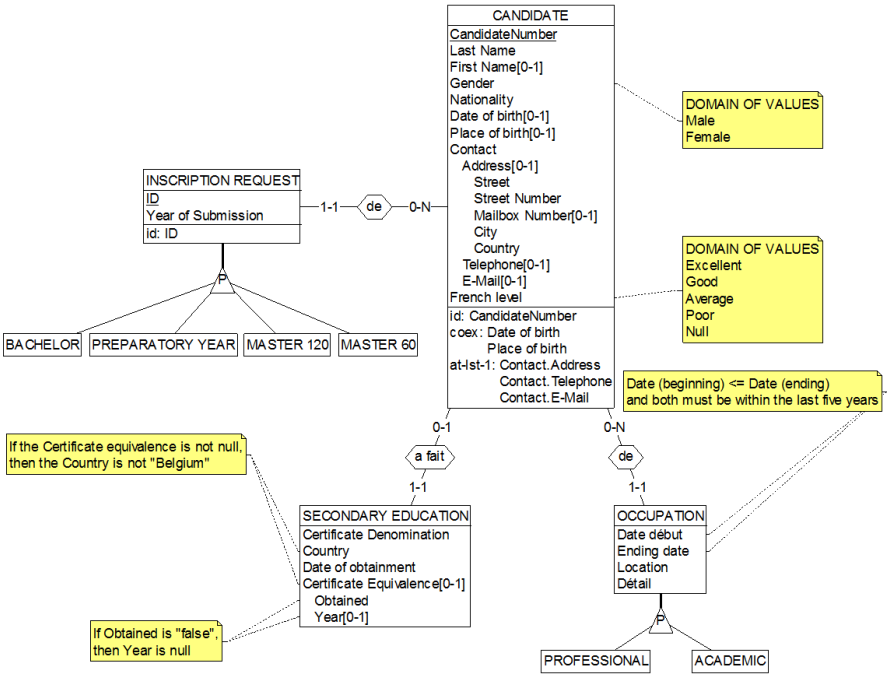


Figure 13.8: The schema corresponding to the domain of the first case study, as conceived by DB1 without seeing the final output schema.

same as the one modelled by DB1, but there is obviously room for technical identifiers and integration between the entity types. Which entity types could therefore be transformed into compound attributes of other entity types, and under which conditions? Could it be when the entity type has no identifier, or no other roles? For instance, `Identity`, `Contact` and `Spoken Languages` could be transformed into compound attributes of `Candidate Information`, and the attribute `Year of submission` could simply be moved into `Inscription request`. In the contrary, `Curriculum` and `Secondary education` are relevant enough to be maintained as entity types. This illustrates the semantic arbitration that an analyst can provide, on the contrary of a machine.

13.2 Second case study: An academic event management system

13.2.1 Preparation

Defining the subject

With the future restructuring of responsibilities within the Faculty of Computer Science, EU2 will be fully in charge of the organisation of seminars, symposiums, conferences and other kinds of meetings. The study will therefore focus in designing the preliminary forms that could be used to facilitate the encoding and reuse of information, to handle the various reservations, the planning and so on.

Training

Before the beginning of the sessions, EU2 received the two screencast tutorials explaining how to use the RAINBOW toolkit to draw relevant form-based interfaces. Because of her full agenda, she did not have much time to spend on the tutorial. She mainly suggested the tutorial to be provided with a locale translation.

13.2.2 Session 1: Drawing the forms

The first session focused on drawing the necessary forms to encode new meetings and registration for those meetings, and was organised as follows:

- *Introduction*: the main observer (MO) introduced the participants and their roles, as well as the objectives (drawing the forms supporting the encoding of information for the subject defined by EU2) and the procedure (EU2 and DB1 draw jointly while OB2 and MO observe) for the current session (10 minutes);
- *Recapitulation*: the participants recapped and discussed the subject of the study, and discussed the tutorials (10 minutes);
- *Execution*: the end-user and the analyst executed the tasks associated with the current session using the RAINBOW toolkit while the observers took notes (100 minutes);
- *Individual debriefing*: afterwards, the main observer discussed separately with each other participant to take their impressions and remarks (15 minutes per participant).

In the following, we expose the remarks made by the observers and the participants through the debriefing.

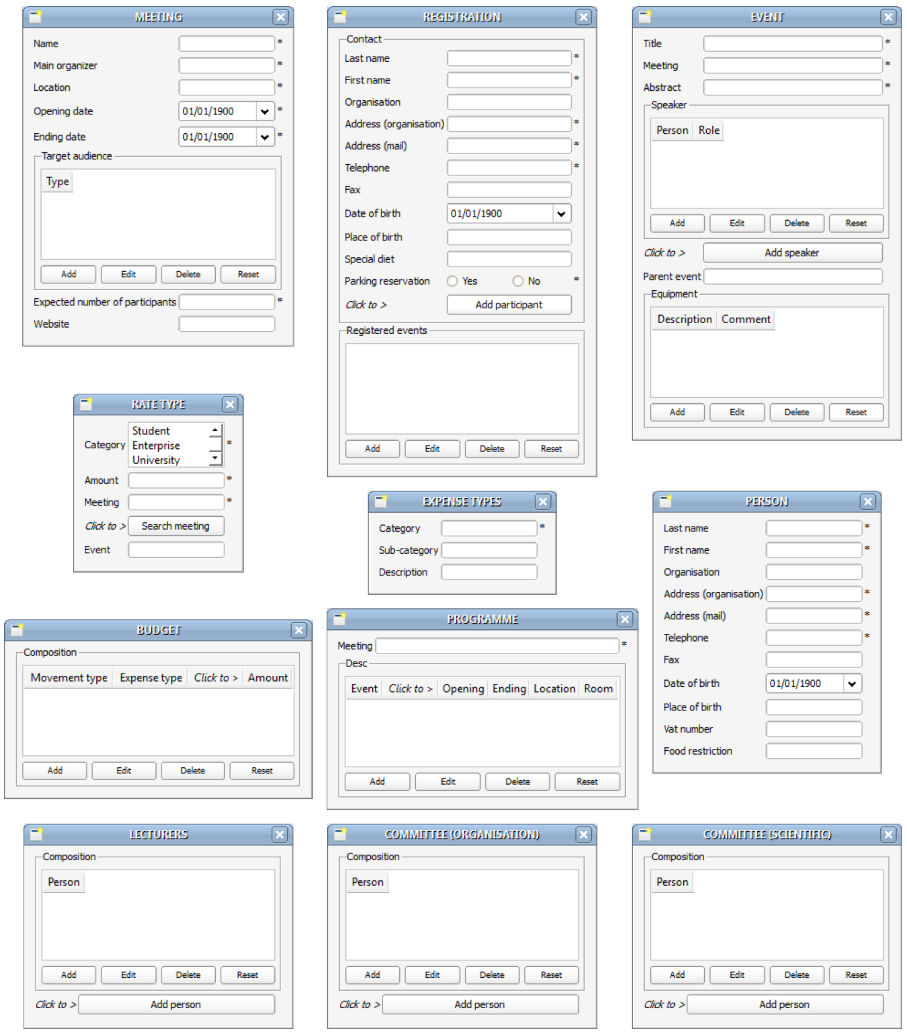


Figure 13.9: The forms drawn by end-user EU2 and analyst DB1 during the first session, and the corresponding raw schema.

General observations

The end-user naturally gave the commands to the analyst and was reluctant to manipulate the rainbow toolkit. It was therefore the analyst who manipulated the toolkit, and he felt more involved and useful this time, while concealing specification difficulties, such as the "parent-dependent" property.

The end-user appreciated that the analyst operated the toolkit, as she was

able to gather her thoughts while he drew the forms. Although she had thought about the problem, she would have appreciated an additional preliminary meeting in order to roughly sketch a first set of forms.

The end-user would have appreciated a projector to view the forms on a large screen rather than watching the little screen of the laptop. The resolution could also be larger, so that all the forms can be visible at once.

Observations on the tool support

There was no particular problem regarding the tool support, as it was operated by the analyst. There were no manipulation mistakes, and the analyst arbitrated the labelling suggestions as they presented themselves. However, the end-user and the analyst did brought to light the following elements:

- the toolkit should support graves, acutes and circumflexes in labels. This coincides with the necessity to have locale versions of the tool support;
- regarding the widgets, it could be interesting to be able to order the options of selections; one can also question the relevance of providing labels for actions, since the buttons already have a label and the actions themselves already have a description;
- regarding the rendering of the widgets, tables could adapt to their content (in terms of columns) and the geometry of the widgets should be preserved whenever they are redrawn;
- it would be nice if the toolkit supported drag-and-drop, to insert new widgets, or to move them from a container to another.

Output of the session

During this first session, EU2 and DB1 produced the forms shown in Fig. 13.9. These forms were automatically adapted into the raw schema of Fig. 13.10, before being transformed into the schema of Fig. 13.11 using the previously defined mapping rules.

From the discussion and drawings of this first session, it appears that a `Meeting` is organised by an organisational and a scientific `Committee` and consists of different `Events`. For each meeting, a `Programme` is defined and a `Budget` is established base on the different possible `Expense types`. The `Lecturers` that attend a meeting need to fill a `Registration`, and may benefit from a `Rate Type` corresponding to their status. Information regarding the different `Persons` involved in meetings should also be stored for possible further reuse. As we can see, there are several main concepts that are already apparent and there seems to be several connections between these concepts.

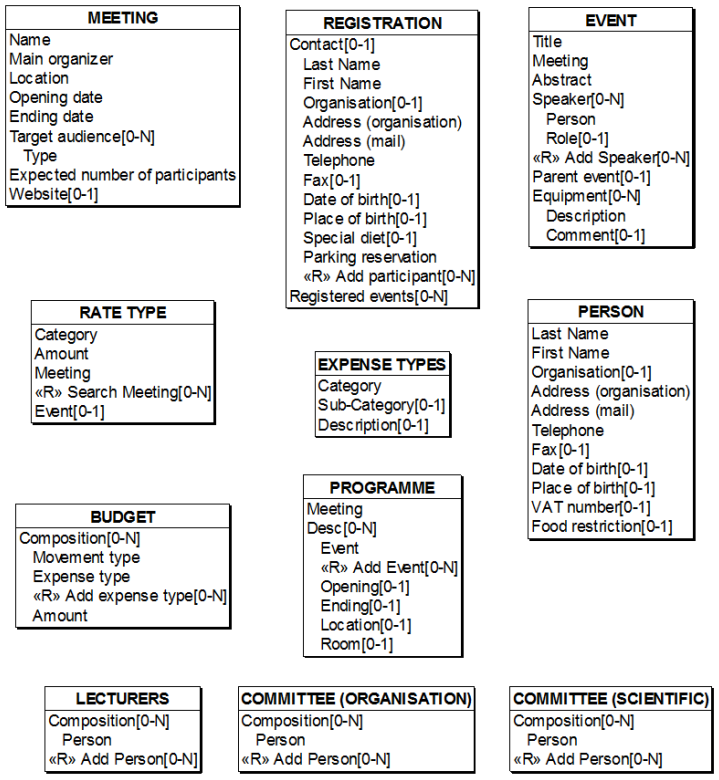


Figure 13.10: The raw schema corresponding to the forms drawn by end-user EU2 and analyst DB1 during the first session (Fig. 13.9).

DB1 and EU2 both felt comfortable with DB1 operating the toolkit under EU2's command. The former felt more useful and handled the building of the forms, while the latter did not feel clumsy with the toolkit and could concentrate on the content. Besides, EU2 appreciated seeing the interface being developing little by little, since it gave her a good overview of the whole project.

13.2.3 Session 2: Analysing the terminology and structure of the forms

This session focused on analysing the terminology and structure of the form, in order to detect any possible ambiguity. The session was organised as follows:

- Recap of the objectives of this session

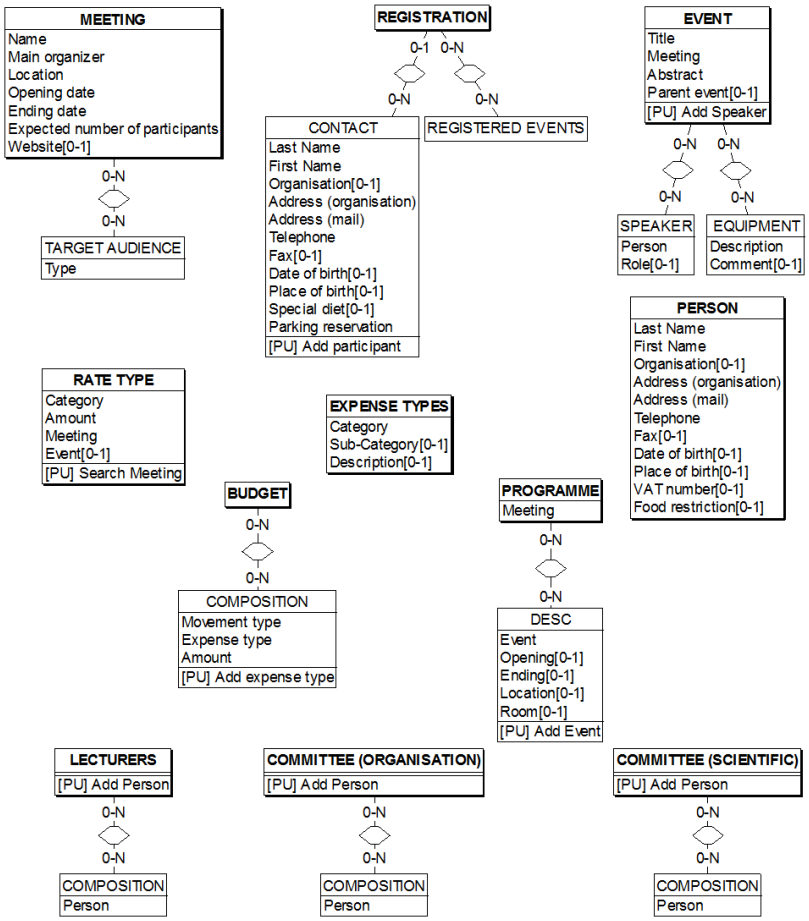


Figure 13.11: The refined schema corresponding to the raw schema of Fig. 13.10.

- Discussion on the previously drawn form, with suggestions of modification.
- Analysis of all the labels: the similar labels have been group in lexically or ontologically similar label clusters; in these cluster, any element that represent the same king of information must be grouped and jointly relabelled (30 minutes).
- Analysis of the similar structures: the forms, tables and fieldsets containing widgets with the same labels are presented for comparison and arbitration; the end-user must explain why such situation occurs (equality, specialisation, union, complementarity or accident).

- *Individual debriefing*: afterwards, the main observer discussed separately with each other participant to take their impressions and remarks (15 minutes per participant).

In the following, we expose the remarks made by the observers and the participants through the debriefing.

Reviewing the interfaces

Based on the notes and discussion from the first session, the analyst suggested modifying the previously drawn form as shown in Fig. 13.12, hence implying the raw schema of Fig. 13.13 and the refined schema of Fig. 13.14:

- Move and restructure the `Committees` and `Lecturers` into their associated Meeting;
- Mention the `Meeting` to be associated with each `Event`, `Registration`, `Budget`, `Programme` and `Rate Type`;
- Replace the column `Person` by the columns `Last Name` and `First Name` in the tables, i.e. use elements that could be detected as referential (`Last Name` and `First Name` can be found in the form `Person`) instead of using an input to refer to a complex form;
- Remove the unnecessary buttons `Add Person` and `Add Speaker`, since they are provided by the table widget.

These suggestions were agreed by the end-user. Subsequently, the updated form was "adapted" using the mapping rules, then EU2 and DB1 proceeded with the execution of their task.

General observations

EU2 had print the forms on paper before the session, and it appeared that it was an efficient way to discuss the forms and annotate them.

Observations on the terminological analysis

The original labelling ambiguities that were detected are presented in Table 13.3. After reviewing these ambiguities, it appeared that there were no labels that still needed to be clarified thanks to the labelling suggestions and the discussions that occurred during the drawing phase. The conflicting elements were therefore different and did not need to be relabelled.

However, this step did lead to loop back to update the forms, especially for the following elements:

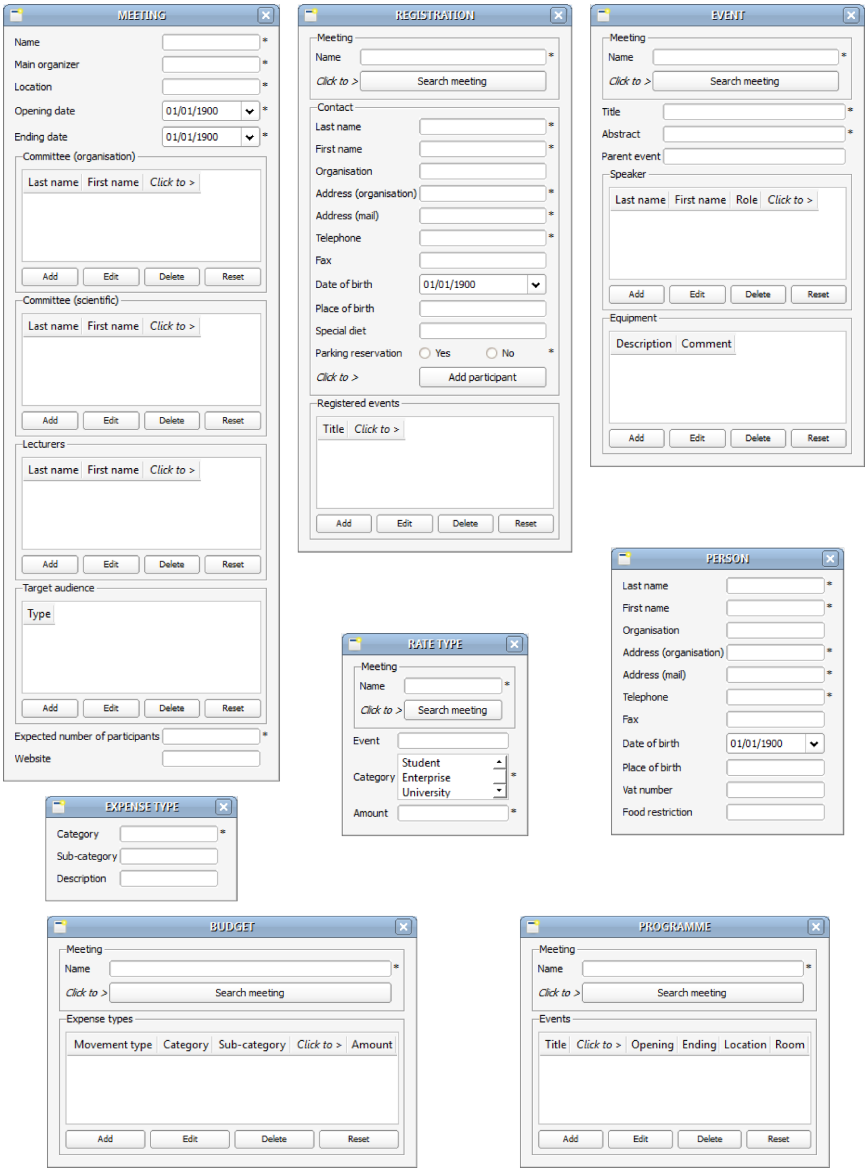


Figure 13.12: The modifications suggested by analyst DB1 to EU2 at the beginning of the second session to replace the original forms (Fig. 13.9).

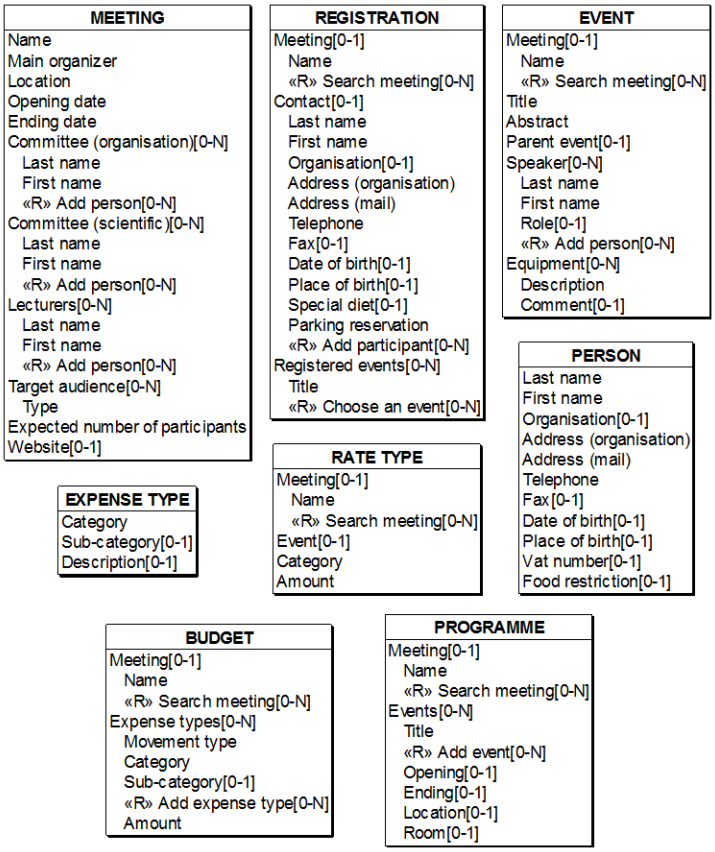


Figure 13.13: The reviewed raw schema corresponding to the reviewed forms of Fig. 13.12.

- Redefinition of the field `Address` into a fieldset;
- Improving the definition of `Event` (by adding a fieldset `Parent event`);
- Improving the definition of `Rate type` (by adding a fieldset `Event`);
- Unifying `Food restriction` and `Special Diet` (under the latter term);
- Improving the order of the elements of the form `Meeting` into a more logical sequence for a better encoding

Observations on the structural analysis

For the sake of conciseness, the original structural ambiguities that were detected and that were not arbitrated as different are presented in Table 13.4.

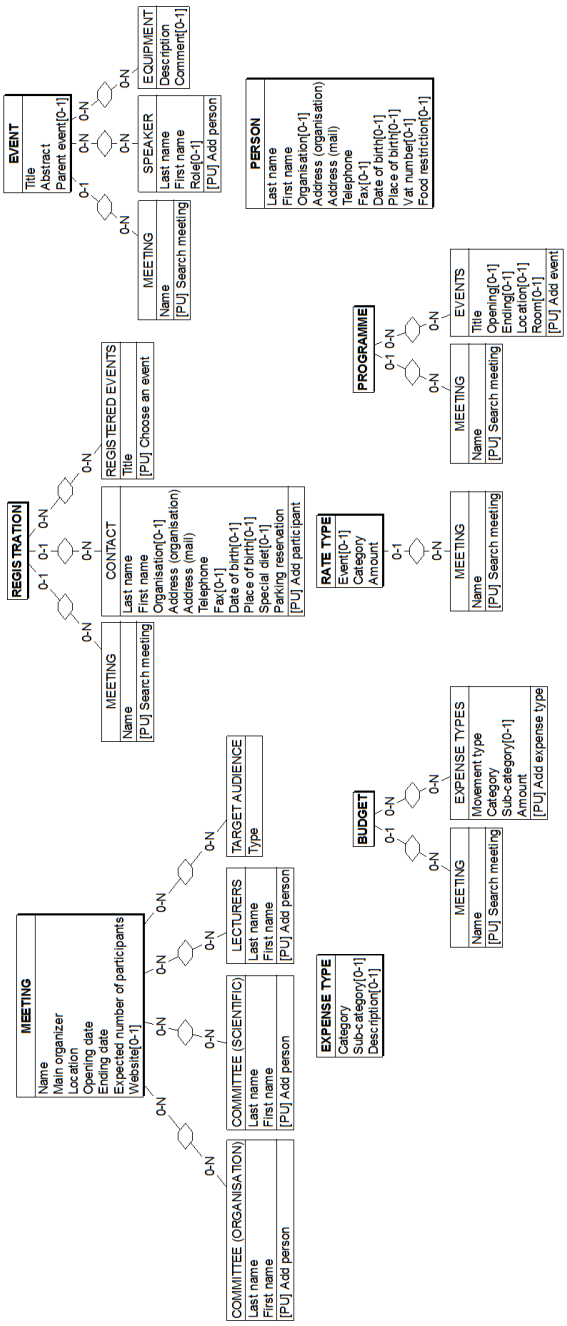


Figure 13.14: The reviewed refined schema corresponding to the reviewed raw schema of Fig. 13.13.

Table 13.3: Labelling ambiguities for session 2 of the second case study.

Ambiguities	**Similar sub groups**
Adresse, Adresse web	/
Catégorie, matériel, sous-categorie	/
Code postal, date de naissance, date debut, date fin, debut, description, evenement, evenements, fin, inscription participant, lieu, lieu de naissance, nom, nombre de participants attendus, organisateur principal, organisme, prenom, regime particulier, role, type, type de frais, type de mouvement, type de tarif, types de frais	/
Comite, localite	/
Num tva, numero telephone	/

In addition, all the `Meeting` containers were equal, and so were all the `Event` containers.

We observed that there was a missing type of relationship, namely "is composed of" that should be added to "equals", "specialises", "unites with", "refers to" and "differs", because of the table widgets.

Output of the session

The form resulting of this session can be seen in Fig. 13.15. After the terminological analysis, the underlying schema was the one of in Fig. 13.16, while after the structural analysis, the underlying schema was the one of in Fig. 13.17.

As we can see, the main redundant concepts that stand out of the schema are the notions of `Meeting`, `Event`, `Person`, `Committee`, `Address` and `Expense Type`. These are the elements that will have to be integrated further on.

As we can see, during this session, there was a major shake-up of the original forms, first because of the suggestions made by the analyst, then accordingly to the discussion raised during the analysis of the terminology. The structural analysis confirmed the intuition that there is a lot of redundancy between the forms of the project.

13.2.4 Session 3: Providing examples and constraints

This session focused on providing and analysing examples to discover explicit and implicit properties of the forms. The session was organised as follows:

- Recap of the objectives of this session (10 minutes);

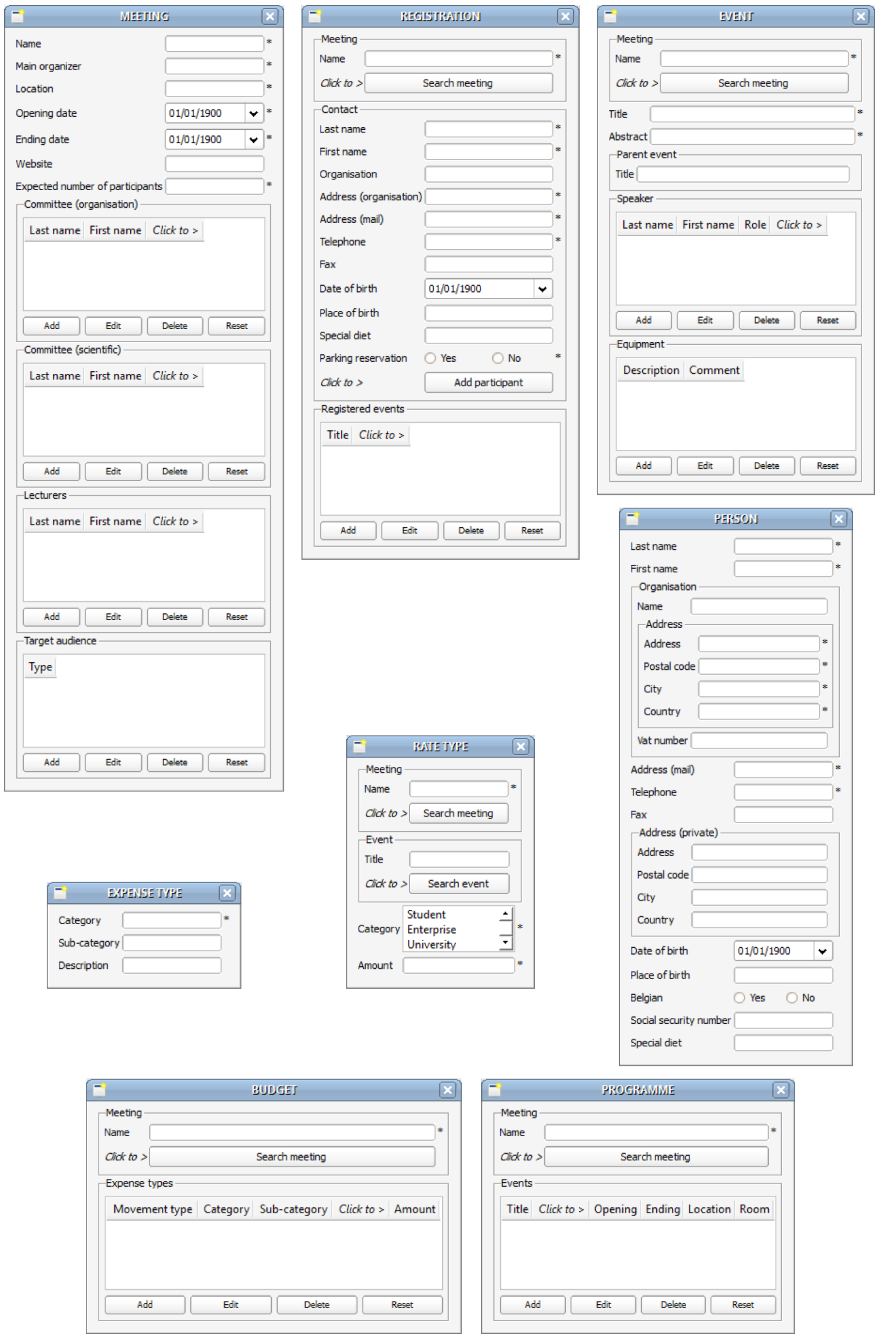


Figure 13.15: The forms at the end of the second session.

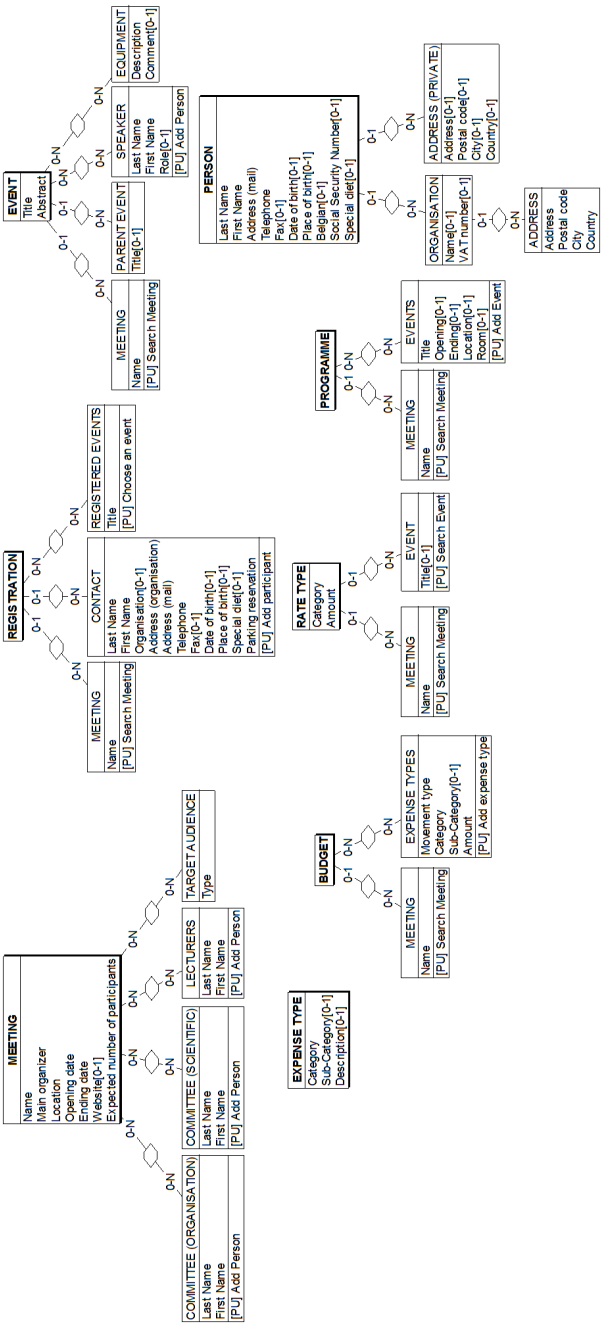


Figure 13.16: The underlying schema of the forms after analysing their terminology during the second session.

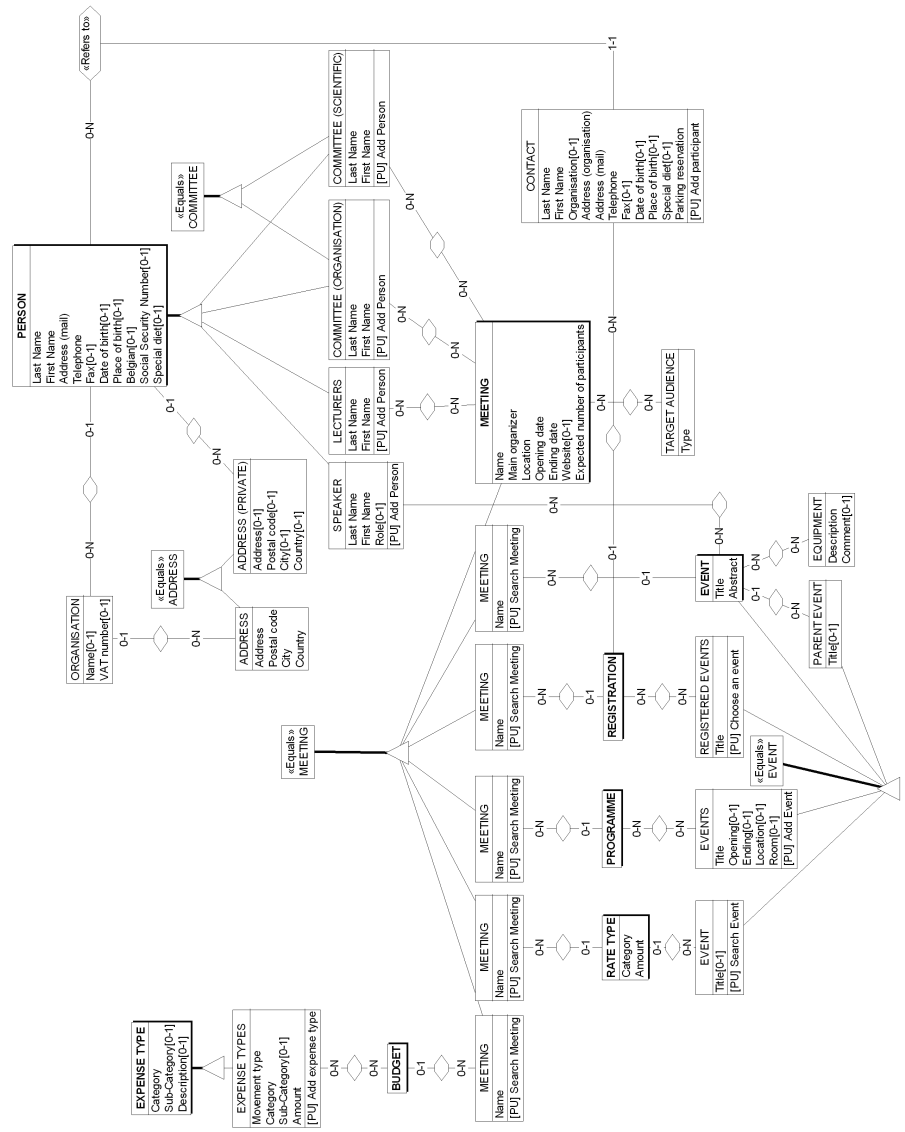


Figure 13.17: The underlying schema of the forms after analysing their structure during the second session.

Table 13.4: Structural ambiguities for session 2 of the second case study.

Ambiguities	Pattern	Decision
Person > Address (private) Person > Organisation > Address	Address, Postal Code, City, Country	Equals
Meeting > Committee (scientific) Meeting > Committee (organisation)	Last Name, First Name	Equals
Meeting > Committee (scientific) Person	Last Name, First Name	Specialises (is composed of)
Meeting > Committee (organisation) Person	Last Name, First Name	Specialises (is composed of)
Meeting > Lecturers Person	Last Name, First Name	Specialises (is composed of)
Registration > Contact Person	Last Name, First Name, Address, Date of birth, Fax, Place of birth, Telephone, Special Diet	Refers to
Event Parent event	Title	Equals
Event Programme > Events	Title	Equals (composes)
Event Registration > Registered events	Title	Equals (composes)
Event > Speaker Person	Last Name, First Name	Specialises (is composed)
Expense type Budget > Expense types	Category, Sub-category	Is specialised (composes)

- Discussion on the previously drawn form, to see if other modifications should be brought(10 minutes);
- Example input and discussion on the properties of the form (165 minutes);
- *Individual debriefing*: afterwards, the main observer discussed separately with each other participant to take their impressions and remarks (15 minutes per participant).

Note that the execution had to be split into two sessions: the first lasted 75 minutes and focused on encoding the examples, and the second lasted 90 minutes to discuss the properties. In the following, we expose the remarks made by the observers and the participants through the debriefing.

Reviewing the interfaces

At this point of the process, EU2 and DB1 did not feel that the form needed to be updated.

General observations

The discussions generated by the examples input led to reconsider the forms `Registration` and `Budget`. In the former, the possibility to specify a funding was added, while in the latter, details on the registration type and the invoice number were added.

During the providing of the examples, it appeared that it would be convenient to be able to copy and paste values from an existing example to another.

The most interesting findings during this session were that:

- If an `Organisation` is mentioned for a `Person`, its `Name` is mandatory;
- A `Person` must systematically provide if he/she is `Belgian` or not (the latter is hence mandatory);
- It is mandatory to specify the `Meeting` associated to an `Event`; moreover, the latter can be identified by the association of the former and its own `Title`;
- A `Meeting` can be identified by its `Name`;
- A `Expense type` has a secondary identifier in the combination of `Category`, `Sub-category` and `Description`;
- A `Rate type` has a secondary identifier in the combination of `Category` and its roles towards `Meeting` and `Event`;

It appeared difficult to express relevant functional dependencies for the other entity types of the schema. Typically, there was no satisfying identifier for a `Person`. It was also difficult to express complex constraints such as the fact that a Belgian `Person` should have a `Social Security Number` while a foreigner should have a `Birth date`.

Since working with the tool raised these issues, the analyst was at least able to note them for further notice, which implies that those information were not lost. Further discussion between EU1 and DB1 confirmed that there should not be any other "hidden" constraint among the given elements.

The discussions also suggested possible improvements that were not considered in our study, but still worthy of interest for eventual further developments:

- How could we handle multiple `Main organizes`?
- For the `Budget`, the `Amount` could be optional and list of usual destinations for trips and travels could be provided;

- It could be interesting to specify if a `Lecturer` should be remunerated or not (through a checkbox for instance);
- Instead of encoding a `Rate type` for each `Category` and for each `Event` of a `Meeting`, a table synthesising the combinations of `Categories` and `Amounts` could prove more appropriate;
- `Expense type` should be encoded before budget;
- A separate `Organisation` form could be useful, with an identifying `VAT number` (when available) and an optional `Department` if relevant, as well as different possible addresses.
- It should be easy to switch from a `Person` to a `Committee` (and vice-versa) so that a user could visualize one and complete the other (or even copy-paste);
- The email and telephone of a `Person` are optional, but there should be at least one contact address.
- In order to allow one `Person` to invite guests, it could be interesting to add a table to `Registration`, enabling additional inscriptions with number of persons, the events, the categories...

We also observed that the existing mechanism for eliciting functional dependencies (FDs) could be improved. The progressive generation of FDs was tedious, because the end-user and the analyst had to discard numerous FDs before getting to the ones they wanted. Instead, it could be interesting to provide a tool to directly define the left and right hand sides of FDs that are trivial.

Output of the session

At this point, the appearance of the forms has slightly changed to reflect the modified cardinalities (Fig. 13.18), but the most visible modifications are visible in underlying schema (Fig. 13.19). It now includes the specified identifiers and has been annotated with the known domain of values and additional information provided by the end-user.

13.2.5 Session 4: Finalising the project

This session focused on discussing the main concepts emerging from the forms, in other words, the ones highlighted during the structural analysis. The session was organised as follows:

- Recap of the objectives of this session;

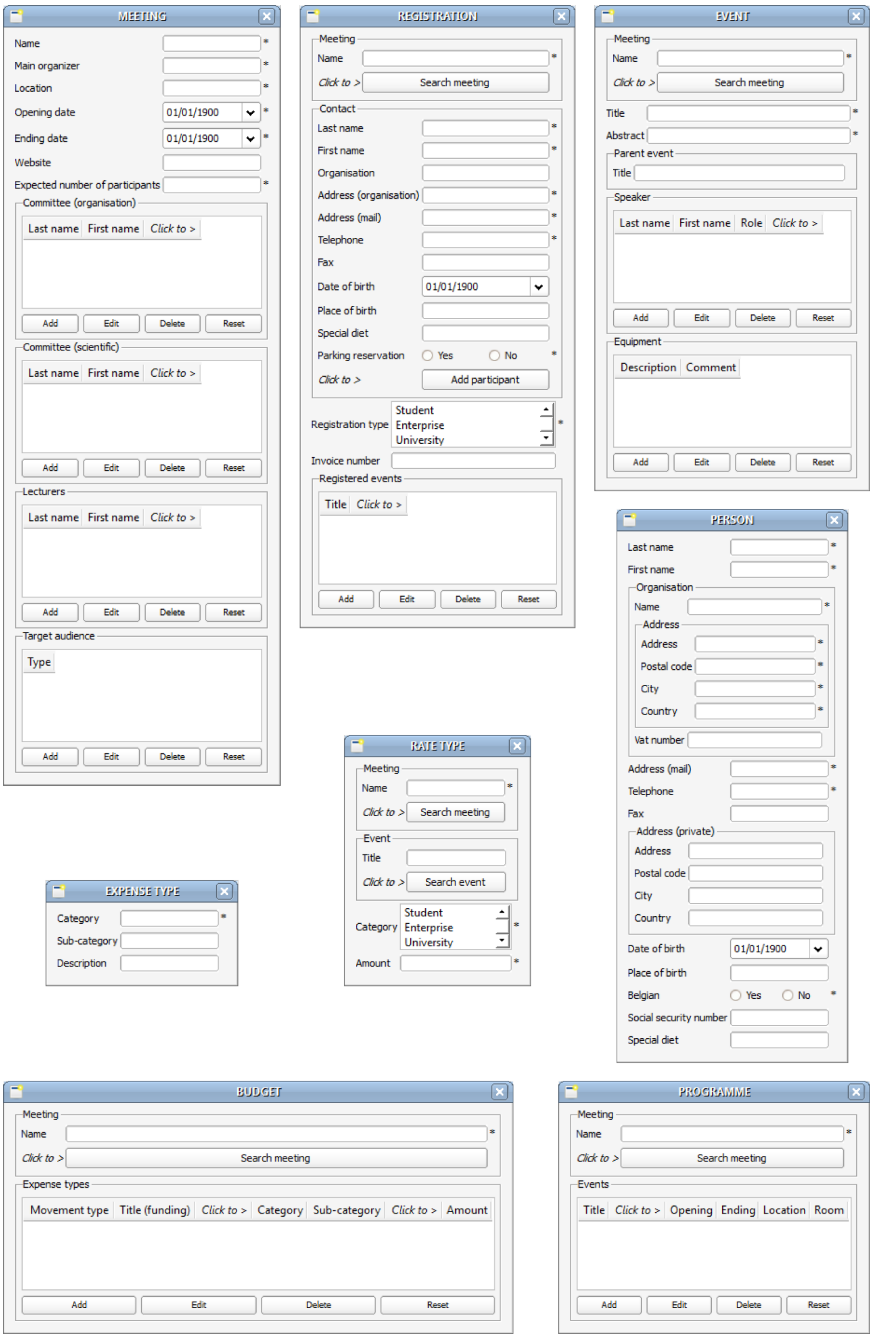


Figure 13.18: The forms at the end of the third session.

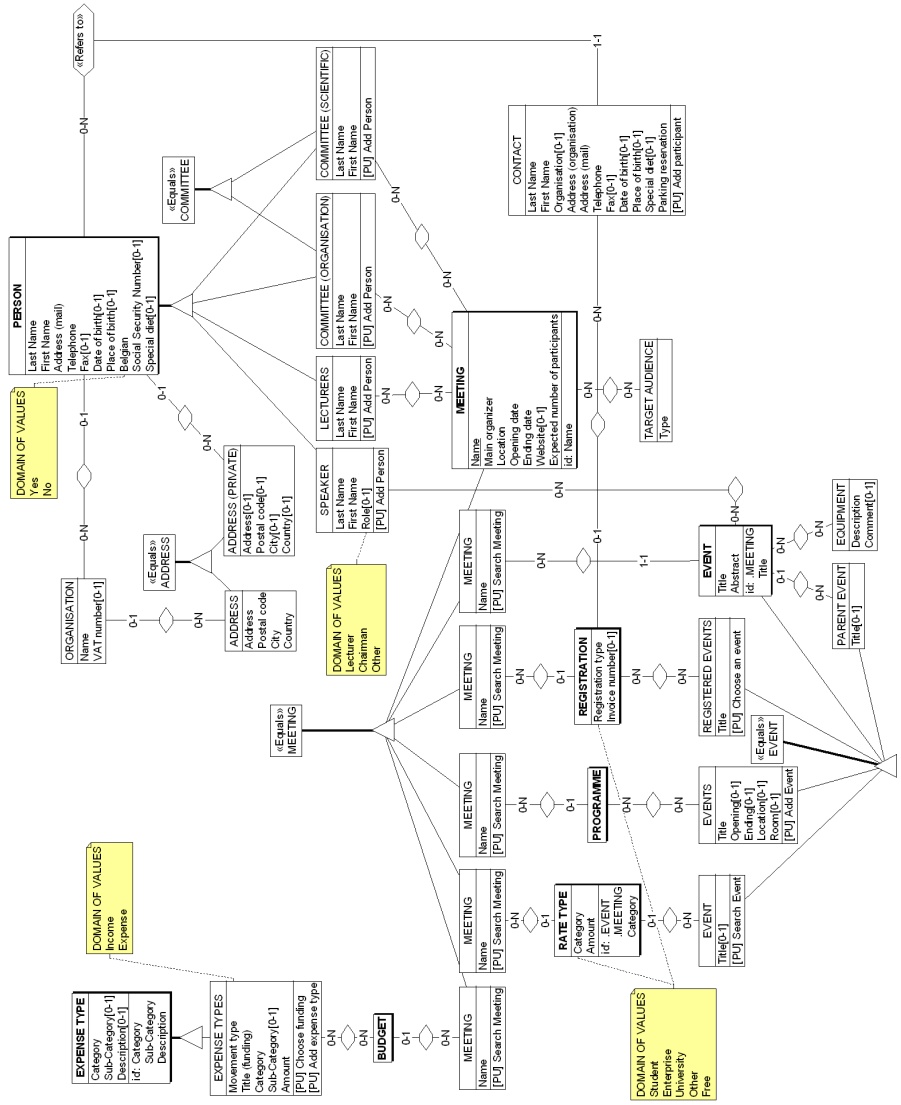


Figure 13.19: The underlying schema of the form at the end of the third session.

- Discussion on the previously drawn form, to see if other modifications should be brought;
- Finalising the project, that is, defining the components (attributes and roles) and their properties, as well as the constraints and dependencies of the main concepts;

- *Individual debriefing*: afterwards, the main observer discussed separately with each other participant to take their impressions and remarks (15 minutes per participant).

In the following, we expose the remarks made by the observers and the participants through the debriefing.

General observations

Here were the observations for each "main" concept:

- `Address`: the discussion on the cardinality of the attributes finally led to defining them all as optional;
- `Meeting`: it was confirmed that the `Name` is identifying and that there should be a mandatory `Main organizer`;
- `Committee`: a `Committee` is indeed composed of `Persons`;
- `Event`: it was confirmed that the `Meeting` was mandatory and part of the identifier with `Title`;
- `Person`: there was no trivial identifier for this concept; besides, similarly to a `Speaker`, the member of a `Committee` has a role (although it is here implicit);
- `Rate type`: it was confirmed that the combination of `Category` and its roles towards `Meeting` and `Event` formed a secondary identifier;

This step was a little confusing for the end-user since it was unclear on what level they were working: the interfaces, their underlying data structures or even the future tables of the database. This implies that special attention should be given to improve this step, for instance by rendering the "main concepts" through form-based interfaces so that the end-user and analyst could see them.

Output of the session

During this session, EU2 and DB1 discussed of the main concepts that were elicited through the investigation step and enriched through the nurture step, leading to the generation of a raw integrated schema of the domain.

At this point, the appearance of the form has not changed, but the underlying schema has (Fig. 13.20). It now includes the specified identifiers and has been annotated with the known domain of values and additional information provided by the end-user. As we can see, there are several "empty" entity types that could be transformed, which will be discussed in the following section.

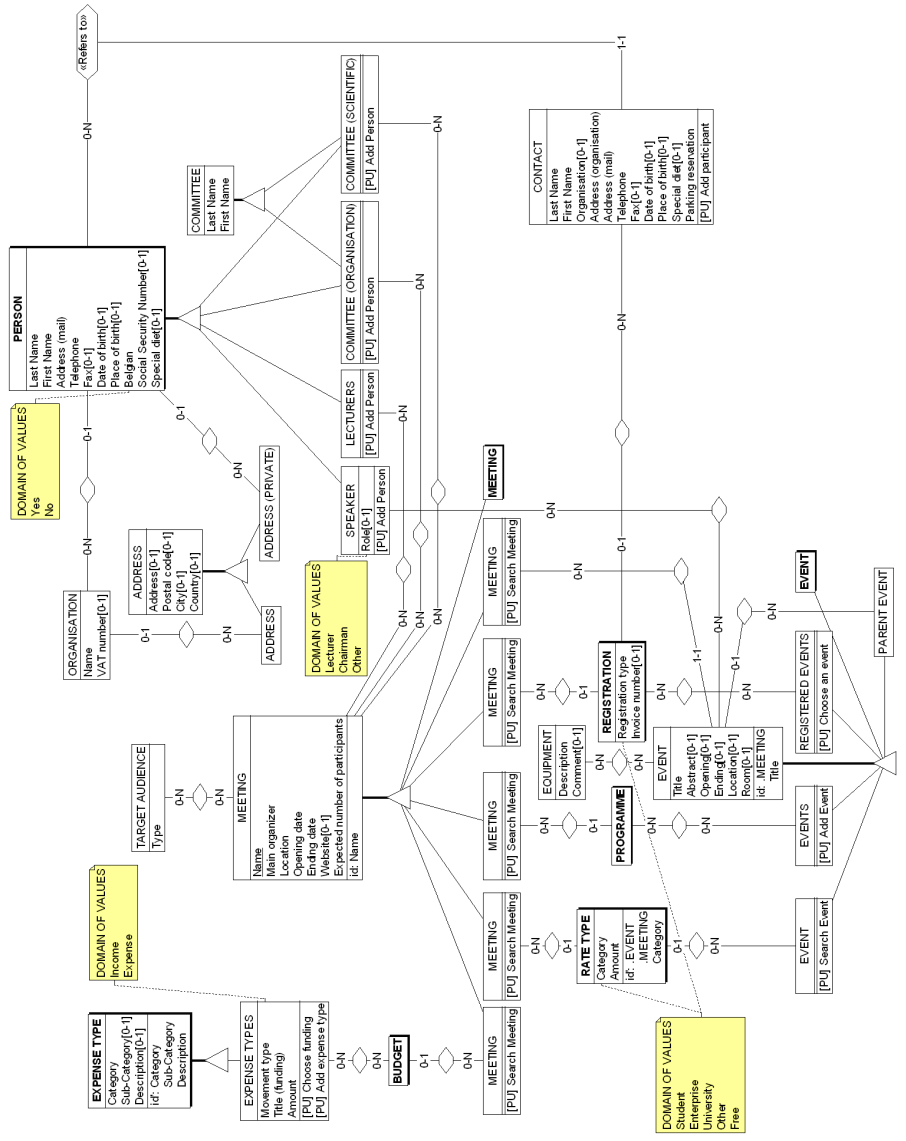


Figure 13.20: The underlying schema of the form at the end of the fourth session.

13.2.6 Discussing the schemas

Characteristics of the subject

This case study was highly interesting because it dealt with a more complex case study requiring complex and multiple forms. There was a lot of redundancies and ambiguities, as well as several interesting constraints to be expressed.

The subjected reviewed by the analyst

Based on the form-based interfaces and the knowledge he gathered during the different steps of the approach, the analyst DB1 drew his own two alternative schemas of the application domain before analysing the schema generated by the RAINBOW toolkit. The result of his modelling can be seen in Fig. 13.21 and Fig. 13.22.

Analysing the generated schemas

We can observe that in this case, the output schema does not have a tree-like structure. It is actually quite complex, with multiples relationship types existing between the different entity types and their hierarchies.

We can also notice that the schema is rather "interface-oriented", with several empty entity types that are subtypes of higher concepts (typically with `Meeting`, `Event`, `Person` and `Address`), while the procedural units might not be relevant at this point.

As previously mentioned, the notion of "composition" could be expressed in a more expressive manner. Indeed, it would have been semantically more appropriate to obtain a `Meeting` having two `Committees`, each of which being composed of `Persons`, while in the current output, a `Committee` is actually a specialised `Person`.

The representation of `Expense Type` and `Expense Types` is also problematic, since the latter could have preferably been labelled as `Movement` and be associated to the former. However, this results from disputable choices during the drawing and structural analysis of the forms. Indeed, the label was probably ill-chosen to begin with, and a posteriori, the pattern `{Category, Sub Category}` that was shared by both entity types rather suggested a complementarity then a specialisation or composition.

Besides, the handling of the "refers to" structural similarity has been left aside, while there are clearly elements that could be integrated, typically for `Person`, `Contact` and `Address`. This kind of structural similarity should also be easily manageable, without forcing the end-user and the analyst to jump back to the drawing step to correct the involved elements.

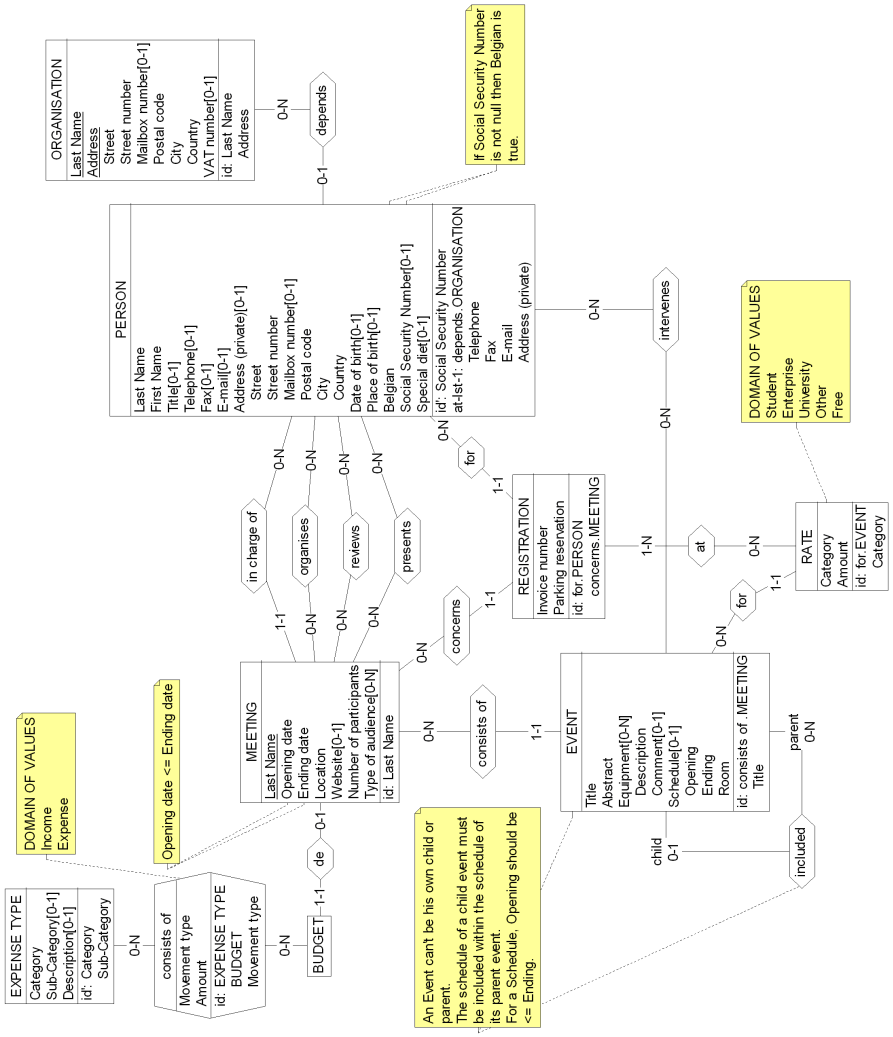


Figure 13.21: The schema corresponding to the domain of the second case study, as conceived by DB1 without seeing the final output schema.

These observations could be dealt with semi-automatically and combined with the implementation of the binding principles that are not provided by the toolkit yet (see Section 11.5). The output of the final session of this case study could therefore be the one illustrated at Fig. 13.23.

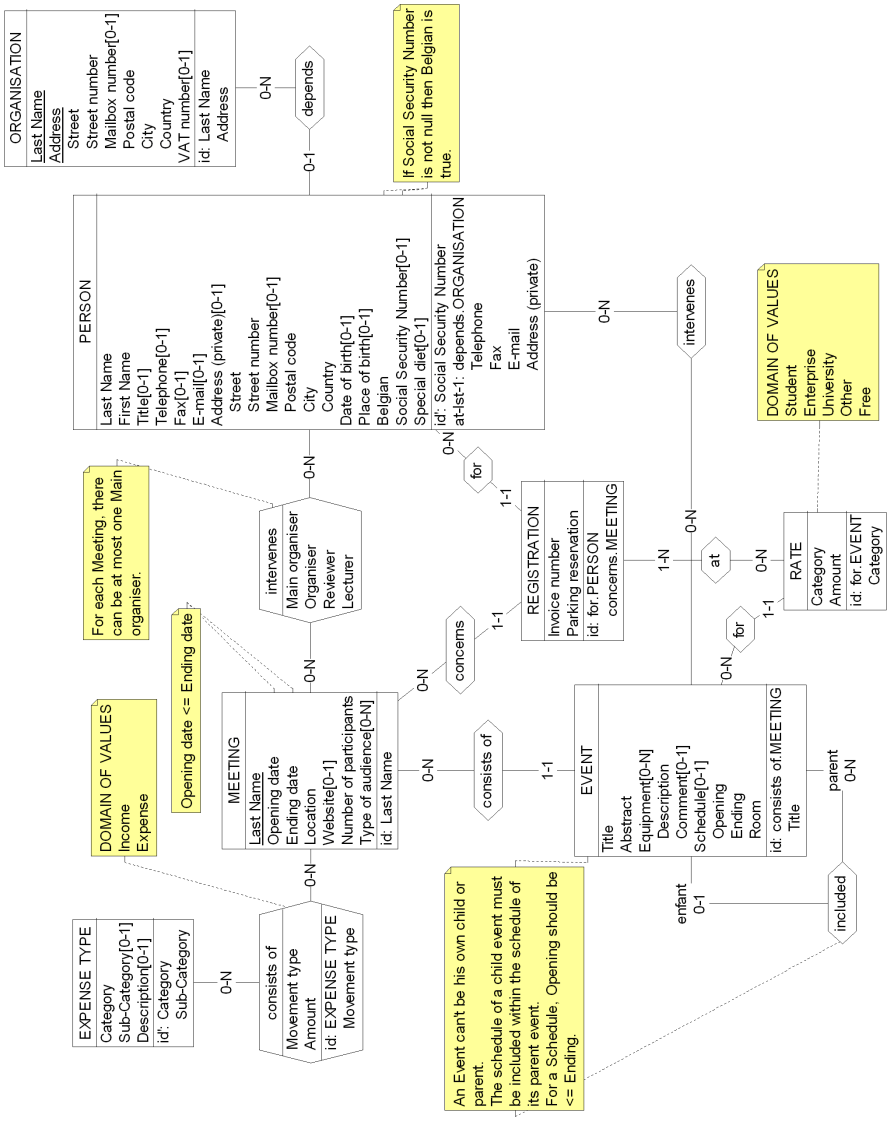


Figure 13.22: An alternative schema corresponding to the domain of the second case study, as conceived by DB1 without seeing the final output schema.

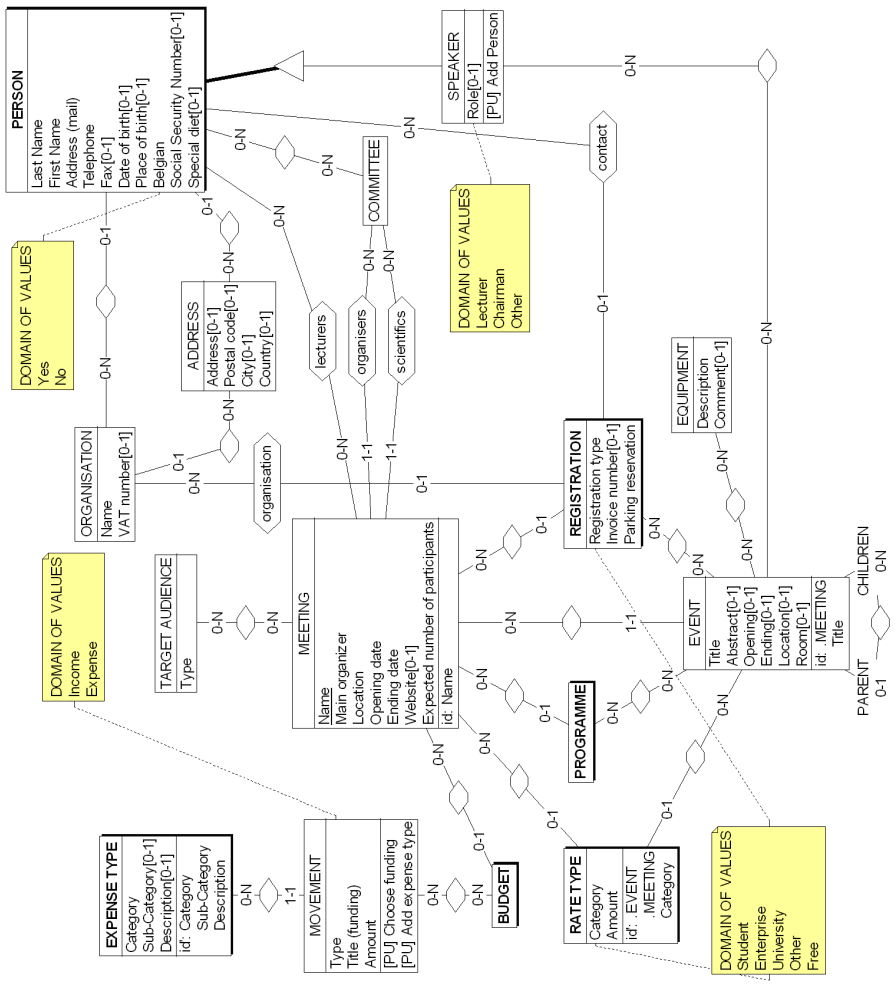


Figure 13.23: The refined schema at the end of the fourth session.

Chapter 14

Discussing the results

In this chapter, we elaborate on the results and observations of this experimentation in the perspective of our two main research questions. Let us recall that the goals of these preliminary case studies were to assess (1) the effectiveness of the RAINBOW approach to express, capture and validate static data requirements, as well as (2) the quality of the conceptual schemas produced using this approach.

14.1 Assessing the effectiveness of the RAINBOW approach

To assess the effectiveness of the RAINBOW approach to express, capture and validate static data requirements from the preliminary case studies, let us first analyse the critical challenges of the approach, then see how they relate to the chosen efficiency criteria.

14.1.1 Expressing concepts through form-based interfaces

Through the drawing step, we wanted then end-users to be able to express concepts by specifying simple encoding screens carrying the data they would need to manipulate in their future form-based application.

The first observation was that the end-users were able to express all their requirements using the drawing tool and the underlying RAINBOW Simplified Form Model (RSFM). For the two studies, the available widgets (forms, fieldsets, tables, inputs, selections and buttons) seemed sufficient, although the elaboration of the forms sometimes called for creativity in the combina-

tion of the available widgets, and occasionally required detailed descriptions to precisely explain the widgets. Still, we also observed phenomena that raised several questions that we will now detail.

For instance, we observed that the end-users often drew single forms to collect multiple informations instead of drawing smaller, simpler forms (i.e. breaking the problem into smaller sub problems). Do the available widgets therefore lead the end-users to draw of single oversized forms, or to create complex structures? It also appeared that the end-users were sometimes challenged by the use of tables, because the latter could only contain simple widgets (i.e. inputs, selections and buttons). Are the tables hence too synthetic? Wouldn't it be problematic, at least visually, to allow them to contain fieldsets and other tables? It is interesting to note that the lack of prior experience in modelling was obviously not a factor in these behaviours.

Besides, we wanted to lead the end-user to focus on the content of the forms rather than their appearance, and subsequently chose an adaptative rendering for the widgets. For instance, selection widgets would automatically switch from radio buttons to checkboxes or a selectable list according to the number of options and the cardinality of the field. However, this behaviour surprised the end-users, and more generally they would have enjoyed at least a minimum of customisation for the rendering of the widgets. Though the forms could be rendered afterwards in more stylish fashions (e.g. with HTML and CSS), could aesthetic considerations lead to a "bad" modelling, just because the end-users want the forms to be prettier? Can the analyst convince them that "it's ok if it is ugly", and can the end-users really agree on that? Or should there therefore be at least a minimum of customisation for the rendering of the widgets? Should the available widgets be presented in an exhaustive manner regarding their cardinality (typically the different combinations of selections)?

Speaking about cardinalities, as previously mentioned, the users must specify how many values could and should at least and at most be provided for each widget. We observed that the end-users often specified widgets as "mandatory", even if they sometimes acknowledged that it would not really be problematic if the given fields were not filled. Could the end-users therefore abusively use this type of cardinality while it is not really necessary? Do they understand the difference between a paper form, which can be submitted even if it is incorrect, and an electronic form which offers immediate acceptance or rejection?

The integration of the "on-the-fly" labelling suggestions in the drawing phase was originally intended to reduce the labelling ambiguities up stream, while being reused during the phase of "Investigation" to ensure the clarification of the labels. However, the end-user and the analyst found that these suggestions were annoying and interfered with the execution of their task. Should it there-

fore become an optional tool for the drawing phase? Should the similarity parameters be adapted to detect fewer ambiguities? Could/should other tools also be parametrised?

14.1.2 Finding and arbitrating terminological ambiguities

The analysis of terminological ambiguities did not yield any significant result. The integration of the labelling suggestions in the drawing phase seemed to be efficient in unifying the terminology of the forms, but in return, it basically made this analysis step useless. It would therefore be interesting to observe what would be the situation if there was no labelling suggestions in the drawing phase.

14.1.3 Finding and arbitrating structural ambiguities

The analysis of structural ambiguities revealed that the notion of "composition" that should be added to the ones of "equality", "specialisation", "unity", "reference" and "difference". Other than that, this step did not cause any concern.

14.1.4 Eliciting constraints and dependencies

As could be expected, this step was the most tedious one of the evaluation. Providing examples was a long task, and it became obvious that it would become difficult to ask for more than 3 to 5 data samples per complex form during the same session without losing the motivation of the participants.

The role of the analyst was critical to filter the interesting technical and existence constraints, as the available amount of possibilities may be high. Eliciting the functional dependencies (FDs) was also challenging, mostly because of the tool support itself. The functionality allowing to generate problematic data samples was never used. However, manipulating the toolkit did lead the end-users and the analysts to discuss thoroughly the possible dependencies and identifiers, including those that could not be expressed using the tool support.

Still, it seems that this step could take advantage of a better tool-supported interaction to order the constraints and dependencies to arbitrate in terms of criticality and likelihood.

14.1.5 Transparently handling integration

This step was only handled in one of the case studies. This step was a little confusing for the end-user since it was unclear on what level they were working: the interfaces, their underlying data structures or even the future tables of

the database. This implies that special attention should be given to improve this step, for instance by rendering the "main concepts" through form-based interfaces so that the end-user and analyst could see them.

14.1.6 Handling user-involvement

Throughout the experimentation, then end-users were receptive to our approach, and embraced the organisation of the experiment. They did not systematically apprehend the underlying objective of each step, but they did understand each task of the process, and thanks to the progressive modifications of their forms, they sensed the developing evolution of their project. We may want to take these positive reactions with caution, because all the participants were willing volunteers and in particular, the end-users were able to define their own subject, which is not often the case in small to medium enterprises that set up a new IT project.

Regarding the assignment of the tasks during the experimentation, we expected the end-users to be more autonomous, especially during the drawing phase. However, during that phase, the end-users were reluctant to operate the toolkit, for various reasons, among which:

- they were afraid not to be able to manipulate it correctly, and therefore give a less-than flattering portrait of themselves to the analyst;
- they felt that the analyst would be swifter and more efficient;
- they needed time to gather their thoughts before drawing each form.

Besides, when the end-users did take the drawing in charge, they recurrently turned to the analyst for advices and explanations. On the other hand, the analyst did not feel very helpful or required for the process when he was not in charge of the drawing, though the end-users felt their presence reassuring. Who should therefore be drawing and who should be assisting? Is the drawing really a job for the end-user? We've been essentially focusing on the end-users involvement, but what about the analyst's own involvement and gratification?

In the end, instead of being a process where the involvement progressively shifted from the end-users to the analyst, the execution of our approach turned out to be a joint development effort. The participants collaborated intimately during each steps, with the analyst serving as an intermediary between the end-users and the toolkit.

To improve this collaboration and make the end-users feel more confidant, a special care should be given to making the environment of the execution reassuring and pleasant. Choosing appropriate settings, such as the location and the equipment, may contribute to securing the adhesion of the participants.

14.1.7 Analysing the efficiency criteria

How do theses observations relate to our efficiency criteria introduced in Section 12.3.1? First of all, there was no major *articulation* problem. The end-users did not seem confused about their task, and their expectations were reasonable. Whenever their demands exceed the scope of the current validation process (typically in terms of graphical rendering or navigation), notes were taken in order to be provided to the persons in charge of the further steps of the project. The end-users did not seem to retain critical information, and were able to express their needs bearing the necessary time to get the grip on the toolkit. They were open to the suggestions made by the analyst, and able to put priorities in their requirements.

The *attitude of the participants* was positive towards the overall process. The end-users were intrigued by this unconventional approach, and enjoyed being intimately involved in the database design course. Seeing the forms progressively evolve made them feel the progression in the elicitation process. The analyst was also rather favourable to this approach, though he felt that he could have been more involved at times, and that some elements were difficult to express using the toolkit.

Most of the static data requirements of the end-users were *expressed using the toolkit*. The information that could not be expressed concerned complex identifiers, transversal dependencies, existence constraints concerning multiple groups of elements, conditional elements and finally, the notion of composition for table elements.

The toolkit induced *discussions* during each step of the process, either for requirements that could or could not be expressed using the toolkit. For the latter, since working with the tool raised these issues, the analyst was at least able to note them for further notice, which implies that those information were not lost.

Regarding the *usability and reliability of the toolkit*, various improvements could be brought, for instance for the overall customisability of the toolkit (parametrizability of tools), the editing of the forms (which could benefit from drag-and-drop features), the elicitation of constraints and dependencies, and so on. Similarly, the *relevance of the elements presented by the toolkit* (similar labels and structures, possible constraints and dependencies) for end-users arbitration may need to be refined in order to present them with more interesting questions.

14.1.8 Assessing the validation protocol

By analysing the results and observations of these experimentations, it appears that the experimentation canvas proved to be valid and relevant, though improvable. Indeed, it notably highlighted that the RAINBOW approach and tool support did help end-users and analysts to communicate static data requirements to each other, while generating a positive response from the participants. Though all the requirements could not be expressed through the toolkit, the latter did serve as a basis for discussion and modifications.

These early results are therefore encouraging, though special care should be given to improve critical aspects such as the assignment of responsibilities, the drawing behaviours, the customisation of the tools and relevance of the elements they highlight. This preliminary validation process also stressed several sensible and interesting phenomenons, such as the emergence of different design styles during the drawing phase, typically regarding the grouping of elements in containers. Such phenomenons will need to be monitored and analysed on a larger scale experimentation

14.2 Assessing the quality of the RAINBOW output

To assess the quality of of the RAINBOW output, we essentially want to analyse whether the analysts were able to gather all the static data requirements necessary to build an appropriate and reliable database. Let us therefore assess how the output schemas compare to the criteria introduced in Section 12.3.2.

14.2.1 Analysing the quality criteria

First of all, the *correctness* of the output schemas was ensured from the beginning, since the mapping rules used in the ADAPT step and the various transformation used in following steps were chosen in order to use only appropriate constructs.

The output schemas did not carry structural and terminological contradictions, which seems to support their *consistency*. This could also be partly expected from the deterministic use of the mapping rules and transformation. However, although it did not occur in the preliminary studies, there could still be structural and terminological variations for what should actually represent the same type of information. Such a phenomenon is difficult to prevent, because of the "Garbage In, Garbage Out" adage, which essentially means that providing incorrect input(s) in a systematic process cannot result in producing correct output(s).

The *completeness* of the schemas was satisfying, since the scope covered the elements that the end-users felt critical, and that they provided themselves the level of detail. However, there were still elements to be added manually, such as technical identifiers or the constraints and dependencies that could not be expressed using the toolkit. Ensuring the completeness is obviously difficult, as in any mono-source approach with the view of a single (or restricted number of) user(s). However, the preliminary analysis of the application domain (as presented in Section 5.3.1) should limit the range of possible omissions in the elicitation process.

The *conciseness* of the schemas could definitely be improved. As observed, there are negligible empty entity types remaining from the integration process, as well as the unresolved "refers to" structural redundancy. These flaws were however expected, since the appropriate mechanisms were not implemented, but as reported, they could be resolved semi-automatically.

The *unambiguity* could also be improved, by handling the conciseness issue, and incorporating the "composition" relation. The remaining ambiguities essentially came from the same "Garbage In, Garbage Out" problem.

The *modifiability* of the schema is not problematic, as long as the toolkit is used to edit it. Advanced mechanisms should however be added if the output schema was to be edited manually before being edited by the toolkit.

The *traceability* of the elements of the schema is ensured, thanks to the unique identifier that is associated with each original form widget and perpetuated in each step of the process. Each element of the schema be therefore be retraced to the original requirements expressed by the end-users.

Regarding the *verifiability* and *testability* of the schema, it obviously can be used to verify if the future application meets the specified static data requirements. Moreover, the original forms can also be used to check if all the necessary fields are present, and if they obey the specified constraints and dependencies.

The *understandability* of the output schemas was also satisfying. The analyst participating to the sessions with the end-users (DB1) felt that they were representative and reasonably similar to the schemas that he had in mind himself. The analyst that did not participate in the sessions (DB2) felt that they were rather comprehensible and expressive. Both agreed that managing the previously mentioned issues would definitely *improve the output schemas*.

14.2.2 Assessing the validation protocol

It appears that the conceptual schemas produced using the RAINBOW approach are of good quality, notably because their content is sensibly the same

as the ones produced by DB1, and that DB2 was able to easily understand them. There are too few examples to assess the representativity of these outputs, yet the validation protocol seems to provide an adequate canvas to observe and assess the quality of the RAINBOW output. These early results are therefore encouraging, though it could be improved semi-automatically with minimal effort to make it less "form-oriented" and redundant.

14.3 Threats to validity

The results of the preliminary studies were promising and tend to give confidence in the feasibility and pertinence of the RAINBOW approach. However, we cannot ignore the numerous threats to validity that surround this evaluation work. First of all, as mentioned earlier, the validation of such a transversal research is intrinsically complex, and would ideally require this approach to be compared to existing ones, based on multiple experimentations led on numerous and different case studies over an extensive time span. We did manage two very different case studies, with different subjects, different representations, different constraints and dependencies... But we only had one end-user in each case, and the same analyst for both projects, which inevitably reduced the potential divergent uses of the approach and toolkit. Studying their use over time with different and multiple participants could in all likelihood reveal different behaviours and other results.

Moreover, we dealt with willing participants who could accommodate their schedule to participate in the experimentation. They were genuinely interested by the project and inclined to provide constructive feedback, probably given that they were able to participate in a project for which they had defined the subject themselves. Real-life projects may not have such favourable settings, and would probably involve more than one end-user and analyst in the process, which may result in less receptive participants. Also, the case studies were led on their own, though they could have served as the starting point of complete software engineering projects (the second case study may actually turn in time into such a project). There was therefore no problematic interference with other requirement engineering or software engineering processes that could normally occur at the same time. This is why, in the next chapters, we will focus on improving the approach and proposing guidelines for a better experimentation canvas, so that one could study the real impact of the RAINBOW approach and compare it to existing approaches.

Part IV

Discussion and Conclusion

In this last part of the dissertation, we discuss the RAINBOW approach and envision possible future works. In particular, Chapter 15 addresses the specificities and merits of the approach and Chapter 16 conversely discusses its limits and improvements that could be considered. Chapter 17 finally concludes this dissertation.

Chapter 15

Specificities of the RAINBOW approach

In this chapter, we present the main specificities of the RAINBOW approach, as a methodology to acquire static data requirements. First of all, we recall that this approach aims at integrating different disciplines in a resolutely user-oriented manner, in order to overcome existing limitations in related researches. Then, we explain how Reverse engineering principles were deviated to perform requirements elicitation. We also expose how this modular and non standard process relies on the transformational paradigm and supports evolution. Finally, after explaining how this model-driven approach can be used in conjunction with other approaches, we argue on the relevance of its output as part of a rich and relevant Software Requirement Specification.

15.1 Integrating different disciplines to overcome existing limitations in related researches

As emerges from this doctoral dissertation, the RAINBOW approach is at the crossroads of different disciplines, each of which deals with specific issues using dedicated methods and techniques. However, as introduced in Part I and developed in Part II, their concerns and subsequent processing can concur for the purpose of bridging the gap between end-users and analysts in order to elicit static data requirements.

Table 15.1: Comparison of existing approaches in prototypical reverse engineering for forward engineering

Method	**FDS/EDDS**	**FLUID**	/	**Click**	**GUAVA**	**AppForge**	**RAINBOW**
Authors	[Choobineh et al., 1992]	[Kösters et al., 1996]	[Rollinson and Roberts, 1998]	[Rode et al., 2005]	[Terwilliger et al., 2006]	[Yang et al., 2008]	Ramdoyal
Tool support (drawing and analysis)	Form Definition System + Expert Database Design System	DIWA + Extended PCTE object management system	Xfig + Prolog + GRL + XVCG	Click	GUAVA framework	AppForge	Rainbow Tool Kit
Prototyping finality	Exploratory, evolutionary	Evolutionary	Exploratory, evolutionary	Evolutionary	Exploratory	Evolutionary	Exploratory, (evolutionary)
Prototype designers	Analysts, end-users	Analysts	Analysts	Analysts, end-users	Analysts	Analysts, end-users	Analysts, end-users
Underlying form model	/	User Interface Analysis (UIA) + User Interface Object (UIO) models	TRIDENT variant	HTML/PHP	GUAVA-tree	HTML	RSFM
Syntactic schema matching	/	/	plurals	/	/	/	orthographic, ontological
Structural schema matching	equality	equality	equality	/	/	equality	equality, specialisation, union, complementarity
Constraints and dependencies	identifiers, FDs	identifiers	identifiers	identifiers	identifiers	identifiers	identifiers, FDs, existence constraints
Examples analysis	static, user-provided and/or generated	dynamic, user-provided	/	/	/	/	static, user-provided and/or generated
Data model	ER	OOA	ERC+ (EER)	Relational Model (MySQL)	Relational model (Natural schema)	ER	GER
Life cyle of the model	linear	linear	linear	linear	linear	linear	cyclic
Target platform	Unrestricted	Unrestricted	Unrestricted	Web-oriented	Unrestricted	Web-oriented	Unrestricted

NB: The symbol "/" means that no details were explicitly provided for the given characteristic.

One of the main achievements of this research was therefore to identify, tailor and integrate principles and techniques coming from the fields of Database Forward Engineering, Database Reverse Engineering, Prototyping and Participatory Design in order to provide this interactive and user-oriented Database Conceptual Analysis approach, and overcome the limitations that were highlighted in Section 3.2 *.

The following sections naturally follow from the decisions that were made to support this integration into a consistent and comprehensive approach. They also detail the contributions of this research with respect to the limitations of existing approaches, which are synthesized in Table 15.1.

15.2 End-users as major stakeholders of the data requirements process

As we have seen, the RAINBOW approach relies on the same principles as the ReQuest framework, which deals with data modelling and the dynamic aspects of the future application, and proved that it is possible to efficiently and swiftly involve end-users in the definition of their needs. However, most laymen end-users were challenged by the task of designing dynamic and rich front-end interfaces supporting the business logic of their future application. Here, we therefore decided to focus specifically on simplifying and improving the static data requirements process, leading the interfaces to appear as a means rather than an end product. In particular, we wanted form-based interfaces to serve as a basis for discussion and joint development, hence using prototyping in an exploratory fashion, though it could be used in an evolutionary approach.

We therefore managed several challenges inherent to this user-centred approach. First of all, to make the development of the interfaces more accessible and to focus the drawing on the substance rather than (ironically) the form, we restricted the available graphical elements to the most commonly used ones, which incidentally also simplifies the mapping rules between the form model and the ER model, and proposed a dedicated tool to support this process.

We also took in account the possible lexical variations that could occur in such an interactive process, which is simply ignored by other similar researches. We therefore offer the possibility to detect and correct on-the-fly many mistakes or deviations in the terminology, or to deal with them later on.

Besides, the interfaces are systematically used to visualise similarities, to input constraints and data samples, so that they can be the referent for the

*It should come as no surprise that the approach overcomes all the identified limitations, as it was precisely designed to do so!

end-users, and their favourite communication means. The end-users therefore interact with the form-based interfaces, while the analyst can also access and edit the underlying data models at any time, as long as he ensures the maintenance of the mapping.

The will to involve intimately end-users into the definition of their needs and the specification of the static data requirements, while managing the satisfaction of all the stakeholders, also places the approach as more suitable for software engineering projects in small to medium size enterprises. Besides, the projects should be themselves small to medium sized, in order to maintain a manageable set of form-based interfaces.

It is interesting to note that though the approach is oriented towards the end-users, the real corner-stone of the RAINBOW processes is the analyst. Indeed, his social and technical skills and knowledge are crucial to manage, assist and guide the end-users in order to perform an enjoyable and effective elicitation process for all the parties involved.

15.3 Using Reverse Engineering for the purpose of Forward Engineering

As exposed in Section 2.2.2, 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. However, using controlled artefacts and monitored processes, our objective is here to "build the truth" rather than "find the truth". In particular, the form-based interfaces are used as a well-defined specification language, as opposed to the usual reverse engineering approach, where the existing screens are obscure artefacts that need to be decrypted. This requires to significantly adapt the usual database Reverse engineering (DBRE) methodology [Hainaut, 2002].

Indeed, as recalled in Figure 15.1 (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) *Conceptualisation*, 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 of RAINBOW approach, as shown in Figure 15.1 (b). Starting from a set of user interfaces ($UI_1, UI_2, \cdots, UI_N$), the physical extraction does not allow one to

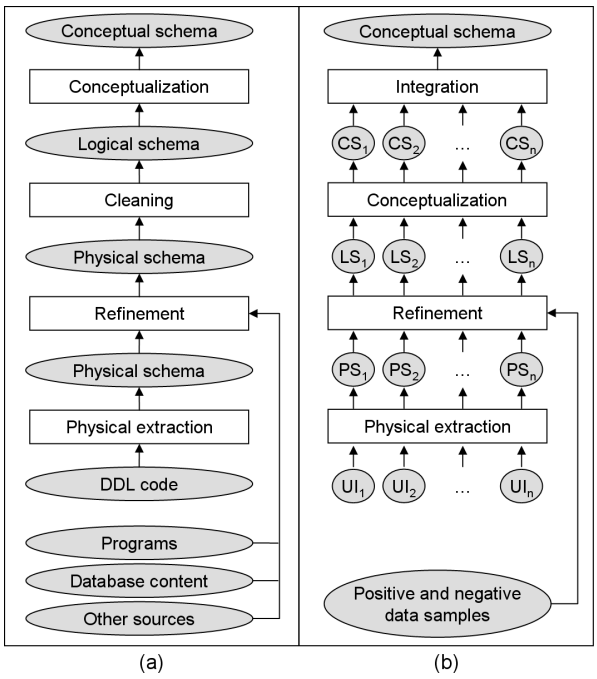


Figure 15.1: (a) Standard database Reverse engineering methodology. (b) Reverse engineering methodology of the RAINBOW approach.

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 artefacts 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 conceptualisation 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 normalised in order to ease the identification of similarities between them. This important process relies on transformation techniques. During the integration phase, the partial conceptual schemas are merged, based on structural and semantic similarity criteria, in order to produce a single complete conceptual schema.

15.4 A modular and non standard view integration process

One of the key assets of the RAINBOW approach is its flexibility, especially regarding the enrichment of the data models. As we have seen, proficient end-users can already provide constraints during the drawing phase. Otherwise, such properties can be directly provided later on, or discovered from a set of data samples provided by end-users. Similarly, the unification of the terminology and structures can also be led during the drawing phase, or during further steps.

This modularity makes the approach suitable for different types of users, ranging from the layman end-user to the advanced database engineer, or from the analyst to the developer. The progressive gathering of elements of integration for further resolution also differs from the standard integration processes.

15.5 A transformational and evolutive approach

The approach also heavily relies on the *transformational engineering* paradigm, according to which most (if not all) Database Engineering processes can be modelled as a chain of schema transformations. Recall that a transformation operator is defined by a rewrite rule that substitutes a target schema construct for a source construct (see Section 2.2.4). The transformations that we use are incremental and preserve the semantics of source constructs in their target counterpart, which ensures the consistency, traceability and reversibility of the specified elements through out the whole approach.

This also favours the evolvability of the specifications produced via the approach. Indeed, our approach is designed to loop if necessary, while storing all the previously provided specifications and decisions. Combined with the traceability of the elements, we can ensure the propagation of any modification in the different steps of our approach.

15.6 An interoperable model-driven approach

The transformational aspect of the approach also highlights that it foremost focuses on developing two main types of models (interface and data), by taking advantage of their connexion while benefiting from the possible use of other types of models (such as tasks models), which places it at the very heart of Model-Driven Engineering (MDE) [Schmidt, 2006]. Furthermore, this approach is interoperable with other MDE approaches, which we can illustrate with two likely circumstances by way of example.

It is notably noteworthy to mention the CAMELEON framework [Calvary et al., 2003], which is a unifying reference framework for developing multi-target user interfaces based on three main steps. First, *ontological models* are defined to describe the problem regarding its *application domain* (concepts and tasks), its expected *context of use* (user, platform, environment) and its possible *adaptation* (evolution and transition). From these models, the *design* phase produces a set of executable user interfaces each targeting specific contexts of use, based on the successive definition of *task-oriented* specifications, *abstract* user interfaces, *concrete* user interfaces (platform-independent)and *final* user interfaces (platform-dependent). Finally, a *run-time* configuration is built from a run-time infrastructure and the user interfaces produced in the design phase, which cooperate to support run-time adaptation. In this context, the RAINBOW approach could be used as part of (or in conjunction with) CAMELEON-compliant approaches, since it addresses the definition of the application domain (the focus being on the concepts) and provides a simple Abstract User Interface model with direct Concrete User Interface counterparts.

As a second illustration, let us consider the OO-Method approach [Pastor et al., 2001], which is built on the basis of OASIS [Pastor et al., 1992], an object-oriented formal specification language for Information Systems, and is notably used by OLIVA NOVA, a software system that generates complete applications from software models [Pastor and Insfrán, 2003]. The OO-Method basically relies on two main modelling components, which are the *conceptual model* and the *execution model*. The conceptual model is divided into four complementary views, namely the *object* view (expressed through UML base diagrams), the *dynamic* view (described through state transition and interaction diagrams), the *functional* view (which classifies the different classes attributes among different types of information patterns) and the *presentation* view (which specifies, through presentation patterns, how the users will interact with the system). The execution model then defines the implementation-dependent features associated with the software representations corresponding to these conceptual modelling constructs. In this context, the RAINBOW approach could notably be used during the definition of the object view to ease the expression of classes and attributes, while reusable interface components could typically be associated to specific presentation patterns.

15.7 A rich and relevant part of a SRS

The output of this process is a set of annotated form-based interfaces and their underlying integrated conceptual schema, as well as their associated playable prototype and ready-to-use database. Compared to other existing approaches,

the resulting conceptual schema is rather rich, since it includes hierarchies, as well as constraints and dependencies. It can also be analysed to generate a thesaurus of the application domain.

Besides, as corroborated by our experimentation (see Section 14.2), this schema constitutes a relevant part of a *Software Requirement Specification* (see Section 2.1.2), since it is consistent, complete, concise, unambiguous, modifiable, traceable, verifiable and testable, bearing an appropriate tool support.

Moreover, the produced elements can effectively be used to share and validate requirements. Indeed, the RAINBOW approach ensures their validation and correction, and these artefacts can be used for further evaluation and reference, while contributing to the forecast of future design and implementation, as well as contractibility.

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 behaviour of the system. In addition, analysing 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 artefacts and even induce more general considerations on the definition of the target information system.

Chapter 16

Possible improvements and future works

In this chapter, we discuss the limits of the RAINBOW approach and its possible subsequent improvements. First, we focus on the current theoretical principles of the approach, and consider possible extensions. Then, we assess the existing tool support and the enhancements that would be welcome step by step. Finally, we recall that the approach would require a wider experimentation scheme, which consequently calls for an improved experimentation canvas.

16.1 Extending the approach

In this chapter, we propose a non exhaustive list of theoretical extensions that could be made to improve the RAINBOW approach, notably based on the narrowing decisions that we took in this doctoral research, as well as the observations and discussions that took place during the experimentation.

16.1.1 Implementing the Objectify and Wander steps

In the scope of this research, we mainly focused on the five first steps of the approach, which were the most challenging given that the generation of applicative components is known to be relatively straightforward, and that the manipulation of a reactive prototype mainly added another level of validation. However, taking the time to formalise these steps would undoubtedly give even more weight to the approach as an integrated end-to-end process.

16.1.2 Incorporating dynamic aspects

We also focused mainly on static data requirements, whereas the analysis of dynamic elements also provides a rich set of specifications that could extend and complement them.

First of all, behavioural modelling could for instance be integrated in the approach by formalising the notion of *formula* that was introduced in the simplified form models in general, and the notion of *action* for buttons in particular. A simple formula language resembling OCL (*Object Constraint Language*, associated to UML) or the ones of spreadsheet programs could for instance be defined, while the most common types of form actions could be classified and made available in the drawing of forms.

Besides, program profiling and comprehension could also be integrated in the Wander *step* to analyse how the end-users effectively use the generated lightweight data manager.

16.1.3 Improving reusability through the drawing support

We concentrated on the use of the RAINBOW approach for independent software engineering projects. However, since the approach is preferably suitable for small to medium size enterprises, the requirements that need to be specified may recurrently be somehow related. Typically, concepts such as "customer" (or "person") and "address" may occur almost systematically from a project to another. Besides, a enterprise could decide to develop other applications reusing an existing database that was developed using the RAINBOW approach. It would therefore be highly recommended to allow the definition and reuse of predefined and reusable structural components, and possibly dedicated ontologies and thesaurus. Such components could also integrate constraints suggestions, example data samples, alternatives terms, etc. Assistance to the end-users could be extended accordingly, e.g. by suggesting such structures on-the-fly.

16.1.4 Refining the terminological and structural analysis

The terminological analysis could also be refined, to avoid reporting irrelevant similarities. This issue remains intrinsically complex, but it would still be possible to improve the results, typically by extending the comparison algorithms. For instance, terms could be compared using multiple string distances, and possibly include alternative string comparison based on the phonetic distance. The term analysis could also detect possible relationship between similar terms based on adjectives, such as order ("first", "second", ...) or prevalence ("main", "alternative"). Regarding the structural analysis, the handling of *compositions*

should be integrated, and mechanisms for characterising components as *referential* could also be explored.

16.1.5 Expanding the analysis of data samples, constraints and dependencies

During this research, we mainly focused on technical, existence and unique constraints, as well as functional dependencies within single entity types. However, it could be valuable to diversify our scope, for instance to also handle multivalued dependencies. Besides, as we mentioned, we reasoned on *valid* (or *positive*) user-provided data samples to highlight possible constraints and dependencies, but it is also possible to consider *invalid* (or *negative*) data samples for that purpose. By exploring and detailing the criteria for invalidity, we could possibly highlight additional constraints or dependencies. Using typical predefined or reusable data samples could also ease the encoding of data samples.

16.2 Improving the current tool support

In this chapter, we propose various adjustments that could be made to improve the execution of the RAINBOW approach through its dedicated toolkit, notably based on the observations and discussions that took place during the experimentation. Let us recall that the toolkit is currently an exploratory and prototypical tool support, and that its limitations result from this stance.

16.2.1 General observations

A transversal improvement concerns the configurability of the tool support, in order to support the participants in a relevant and non intrusive fashion (which we will specify on a case by case basis). Another concern that appears in each step is the possibility to edit the interfaces and display their widgets properties without necessarily looping back and replaying each previous step. Finally, the toolkit should be adaptable to different languages, which would imply, among others, to extend the authorised character set in labels.

16.2.2 Drawing

Here are some possible improvements for the drawing step, which could be implemented and tested:

- the toolkit could provide a drag-and-drop feature to insert and move widgets;

- widgets could be "transformable" into another one, e.g. an `input` into a `selection`;
- the integrated label analyser should be parametrisable and easy to turn on and off, since it unnecessarily triggered too often;
- the integrated label analyser could be integrated as a silent dockable element of the toolkit instead of popping up whenever there is a lexical or ontological ambiguity;
- the available widgets could be presented in an exhaustive manner regarding their cardinality (typically the different combinations of selections);
- a dedicated widget for referential elements could be introduced;
- `tables` should have the (manual) possibility to adapt to their content;
- the options of a `selection` should be orderable;
- the actions of a `button` could be connectable to a set of simple but effective actions (such as navigating between forms, choosing an existing data sample, ...);
- ...

16.2.3 Investigate

The layout of the terminological and structural tools should be improved. To do so, alternative arrangements could be submitted to end-users in order to get their feedback: for instance, groups of similar labels could be presented one by one, instead of being shown all at the same time.

The label analyser produced large groups of similar labels, which was rather gruelling to process. The elicitation and grouping of lexically or ontologically similar labels could therefore be tested with other parameters and/or strategies. For instance, we could handle similar labels as a graph of interconnected labels instead of independent terminologically similar subsets. Each label would be a node, and would be connected to other nodes through edges characterising their computed terminological similarity, the latter having to be validated by the end-users.

Besides, the handling of the "composition" and "complementarity" structural similarities should also be easily manageable, without forcing the end-user and the analyst to jump back to the drawing step to correct the involved elements: the toolkit should instead include transformation tools to process/update the forms and their underlying data model. Also, the toolkit should include tools to add the "missing" form for containers that share a "union" similarity.

16.2.4 Nurture

The nurturing step seems to be the most fastidious one of the process, in great part because providing examples demands a great effort and arbitrating the constraints is challenging. In order to facilitate the encoding of examples, the end-users could rely on existing data samples and (if they want) ask the analyst to encode them under their supervision. Also, it should be possible to easily copy/paste values from existing data samples

Regarding the arbitration of constraints and dependencies, it might be profitable to order them in terms of criticality and likelihood. Instead of using summary tables for the constraints, the information could be structured differently, for instance by generating "readable" questions (e.g. "Is this element mandatory?") and hiding the unquestionable elements.

For existence constraints and functional dependencies, it would be nice to be able to directly define their components, instead of having to go through example input and/or manual discard. Besides, complex identifiers, transversal dependencies and conditional elements should also be handled.

16.2.5 Bind

This step is currently quite abstract, since we do not directly work with the existing forms. To stay consistent with the approach, we could precisely render the "main concepts" through form-based interfaces so that the end-users and analysts could see them and compare them with the other forms they drew. The generated form could be added to the project, and the related existing forms could be annotated to specify this reference. Also, an integration assistant should be provided to semi-automatically process the remaining elements, such as empty entity types.

16.2.6 Objectify and Wander

As previously explained, the generation and integration of applicative components into a "playable" prototype and the testing of that prototype by the end-users have been deliberately left aside in the context of this doctoral research, but it would definitely be interesting to implement them once they are theoretically explored in depth.

16.3 Pursuing the experimentation based on an improved canvas

We believe that the protocol that we presented in Section 12.4 was appropriate to assess the efficiency of the RAINBOW approach and the quality of its output. In this chapter, we therefore suggest to extend this experimentation canvas so that it could be used in a wider experimentation endeavour, bearing in mind the *Participant-Observer* and *Brainstorming/Focus group* principles. In the following, we will hence explain how to prepare further case studies and how to apply and review the RAINBOW approach, before comparing it to other approaches.

16.3.1 Preparing the experimentation

The inevitable restriction is that the chosen software engineering projects must target form-based applications for small to medium sized companies, though it could also be interesting to study how relevant the RAINBOW approach could prove for other types of applications. Additionally, it would be interesting to have a wide range of application domains, to maximize the possible modelling challenges.

All the participants should be familiar with form-based human-computer interactions, such as web forms, and the analysts should be familiar with (static) data modelling. Before starting the apply the RAINBOW approach, the analysts in charge of the (static) data modelling, as well as the observers, should meet with as many of the stakeholders as possible, in order to get the big picture and start thinking about the subject. If there are too many potential end-users, the selection of the participants should be done carefully in order to preserve their sensibility.

All the participants should subsequently receive a general explanation on the RAINBOW approach, the organisation of the sessions and a special training to use the toolkit. In addition to the screencasts, training sessions could be organised, and a sandbox version of the toolkit, including tutorials and examples, should be available for individual testing.

Once the participants are properly trained, the analysts need to define the experimentation settings with the end-users in other to maximise their comfort and willingness. This includes choosing the schedule and location of the sessions, the equipment that will be used, and how multiple end-users will participate (jointly, separately, alternatively, ...).

16.3.2 Applying the RAINBOW approach

The implementation of the RAINBOW approach relies on the joint development of the conceptual schema of the application domain, which will in turn lead the implementation of the database. To perform this process, we advocate to keep the four main interactive steps of the approach, namely Represent, Investigate, Nurture and Bind, by planing one assignment for each of them. Each assignment should be organised in sessions of 60 minutes at most, in order to keep the focus and interest of the participants. If the future evolution of the toolkit supports the Objectify step, the interactive Wander step could be included in the experimentation as a fifth session.

Each session should be organised as follows:

- *Introduction*: recall the previous steps and present the main objectives of the current session;
- *Recapitulation*: discuss the previous steps and the possible elements that remained unclear or that should be reworked;
- *Execution*: execute the tasks associated with the current session using the RAINBOW toolkit while the observers took notes;
- *Individual debriefing*: discuss separately with each other participant to take their impressions and remarks.

In this first assignment, the end-users and the analysts must draw and edit forms that would allow them to accomplish usual tasks of the future application project, with a special attention to encoding forms. They need to focus on the terminology and specification of the forms rather than their layout and general appearance. It could be interesting to push forward the use of labels, for instance by asking end-users to provide the singular *and* plural variations of the labels they use, typically when using tables.

In order to study how the end-users react to the tool support and handle the responsibility of the drawing, they should initially be asked to operate the toolkit to draw the forms. However, if they feel uncomfortable with this task, they could agree to delegate it to the analysts.

During that assignment, the observers should be attentive to the following elements:

- the drawing behaviour, that is, how the participants use the available widgets to represent different types of information and requirements;
- the articulation problems that occur (as presented in Section 2.1), and in which circumstances;
- the information that could and could not be expressed using the toolkit;
- the discussions that were induced by the approach and toolkit;

- the usability and the reliability of the toolkit;
- the behaviour of the label analyser and the relevance of its suggestions, if it was activated.

At the end of each session, a screenshot of the forms should be taken (and possibly printed) and provided to the participants, so that they can continue to think about the project until the next session. In particular, the analyst should analyse the labels and structure of the widgets to detect possible alternative representations. At the beginning of the next session, the participants can then discuss the possible improvements of the current forms.

In the second assignment, the participants must analyse and arbitrate the terminology terminological and structural ambiguities remaining in their forms. At this point, it appears that the analyst should operate the toolkit in order to serve as an intermediary with the end-users.

It is important for the end-users to understand that, ideally, widgets referring to the same concept should bear the same label, and that widgets referring to different concepts should bear different labels. Likewise, the similarity between two containers should be appropriately chosen among:

- *equality* for containers representing the same concept;
- *specialisation* when one of the concepts specialises the other;
- *union* for containers representing specialisation of a more generic concept that is not explicitly expressed through the forms;
- *complementarity* when one of the concepts actually refers to the other;
- *composition* when one of the concepts (expressed through a table) consists of multiple instance of the other;
- *difference* for different concepts.

During that assignment, the observers should be attentive to the following elements:

- the relevance of the generated sets of similar labels;
- the impact of the possible use of the label analyser during the drawing on the elements arbitrated during this assignment;
- the possible trends in the automatically generated structural ambiguities;
- the behaviour during the arbitration of structural ambiguities;
- the articulation problems that occur, and in which circumstances;
- the information that could and could not be expressed using the toolkit;
- the discussions that were induced by the approach and toolkit;
- the usability and the reliability of the toolkit.

In the third assignment, the participants must provide a set of examples, then examine the technical constraints, the existence constraints, the functional dependencies and the possible identifiers associated with each form and its elements. Since this assignment can be extremely time-consuming, multiple sessions may have to be planned to encode examples and elicit constraints and dependencies.

However, though gathering examples is important to generate possible constraints and dependencies, the analysts should cut directly to what he feels to be the most likely relevant ones. Discussing with the end-users based on the submitted forms and data samples may speed up the validation process.

During that assignment, the observers should be attentive to the following elements:

- the variety and relevance of the submitted examples;
- the number of examples that seem to be necessary to obtain valuable suggestions, according to the structure of a given form;
- the number of corrections that had to be made regarding technical constraints (typically regarding the cardinality);
- the articulation problems that occur, and in which circumstances;
- the information that could and could not be expressed using the toolkit;
- the discussions that were induced by the approach and toolkit;
- the usability and the reliability of the toolkit.

In the fourth assignment, the participants must arbitrate the properties of the top-level concepts that were elicited through the previous steps, that is, their attributes and associated technical constraints, existence constraints, functional dependencies and possible identifiers.

It is important for the end-users to understand that they should treat these top-level concepts as forms that should aggregate all the information shared by their "sub forms".

During that assignment, the observers should be attentive to the following elements:

- the articulation problems that occur, and in which circumstances;
- the information that could and could not be expressed using the toolkit;
- the discussions that were induced by the approach and toolkit;
- the usability and the reliability of the toolkit.

16.3.3 Reviewing the experiment and comparing the approach to existing approaches

To assess the effectiveness of the RAINBOW approach, the observers and analysts should analyse the observations taken during the different assignments and sessions. Let us recall the *efficiency-related elements* they need to be attentive to:

- the possible *articulation* problems that were presented in Section 2.1, namely: confusion, improper expectations, difficult or unclear articulation, inappropriate prioritisation;
- the attitude and satisfaction of the participants regarding the methodology (and how they can possibly compare them to other approaches);
- the information that could and could not be expressed using the toolkit;
- the discussions that were induced by the approach and toolkit for the requirements that could and could not be expressed using the toolkit;
- the ease of use and reliability of the toolkit;
- the relevance of the elements presented by the toolkit (similar labels and structures, possible constraints and dependencies) for end-users arbitration.

To assess the quality of the RAINBOW output, let us recall the *quality-related criteria* that the analysts need to analyse:

- *correctness*: does the schema use appropriate constructs?
- *consistency*: is the schema free of contradictions?
- *completeness*: does the schema cover (exactly) all the aspects necessary to conceive the future database of the software engineering project (*scope*), and is it detailed enough (*level of details*)?
- *conciseness*: is the schema free of redundancies?
- *unambiguity*: are there elements of the schema that are still unclear or disputable?
- *modifiability*: can the schema be updated easily ?
- *traceability*: can each element of the schema be retraced to the original requirements expressed by the end-users?
- *verifiability*: can the schema be used to verify that the software meets the requirements?
- *testability*: can pass/fail or quantitative assessment criteria can be derived from the schema?

Besides, they also need to analyse the following *practical issues*:

- Does the approach help the analyst to understand the application domain, whether he was part of the experimentation or not?
- What could and should be done to improve the output schemas?

However, as we have seen, an additional effort could be made to define a more systematic evaluation of the experiments. This would imply reviewing evaluation techniques used in comparable existing approaches over time, as well as Requirements Engineering in general. Synthesising the main evaluation criteria and comparing the values and results for each of them could enable at least a theoretical comparison of these approaches. By moreover asking analysts to participate in different projects using the RAINBOW approach or not, we could get more practical feedback on the flaws, advantages and possible improvements of the approach.

Besides, it would also be interesting to study the evolution aspects of the approach. While we already consider the possibility to "loop" during the steps of the approach as long as we are in the conceptual design, what would be the situation if we needed to edit a working database produced using the approach?

Chapter 17

Conclusion

As this dissertation comes to an end, let us take the time to recall the different stages we went through during this endeavour. All started at the carrefour of Requirements Engineering, Database Engineering and Human-Computer Interfaces, where we wondered how we could combine these disciplines to support the elicitation of static database requirements in the context of Software Engineering.

From this initial existential questioning, we started by examining the context of this research area. We notably investigated how the Software Crisis progressively led to the emergence of Requirements Engineering as a cornerstone of Software Engineering, and presented different aspects of this field. We then focused on Data Engineering, which aims at accurately eliciting and validating data user requirements to help build a reliable documentation of the application domain. In particular, we took a special interest in the phase of conceptual design, which seeks to express user requirements into a conceptual schema, based on data models such as the GER, which are not easily accessible to the laymen, but offers the advantages of transformational approaches. It also appeared that several techniques to acquire data requirements do exist, but that they do not involve actively end-users. Providing a better requirements acquisition process for Database Engineering hence implied bridging this communication gap between end-users and analysts.

Incidentally, we realised that Prototyping was precisely a technique that had proved efficient to elicit and validate requirements, though prototypes are still mainly designed by analysts rather than the end-users, and therefore appear as a one-way communication channel. Nevertheless, form-based interfaces

especially fitted the purpose of transparently expressing formal requirements, which could be used in combination with the principles of data reverse engineering. The few related researches on this domain were reviewed and their limitations exposed, among which the lack of user involvement and adequate tool support to help them focus on the information content of the forms, the assumption that labels are used consistently through out a set of different forms, the non systematic use of data samples to elicit constraints (when the latter are available), the lack of validation on final integrated data models, and the absence of evolutionary perspectives.

We hence naturally wondered about the perspective of designing our own approach to reverse engineer prototypical user-drawn form-based interfaces in order to perform an interactive conceptual analysis. We consequently presented several key problems inherent to the different disciplines that would need to interoperate in order to perform such a process. Regarding Database forward engineering, the main challenges concerned the elicitation of ambiguous elements needing arbitration, in order to prepare the integration of multiple data models and the subsequent generation of applicative components. In particular, we presented String Metrics and Ontologies to discover terminological ambiguities, Tree mining algorithms and Formal concept analysis (FCA) to elicit structural redundancies, and the application of induction, dependency discovery algorithms and FCA on data samples to uncover constraints and dependencies.

We also mentioned traditional view integration strategies to manage schema integration, as well as transformations and CASE tools to generate applicative components. Regarding Database reverse engineering, we pointed out that static and dynamic analysis of forms could be used to extract a set of raw data models from a set of form-based interfaces. As for Prototyping, we addressed the importance of choosing an appropriate User Interface Description Language and an adequate tool-support to express and validate concepts through form-based interfaces, then generate a playable form-based prototype from an existing conceptual schema. We furthermore insisted that the proposed techniques and strategies needed to be tailored in order to promote user-involvement through interactivity, and lead to an integrated and consistent elicitation process.

Subsequently, we presented and detailed the principles and processes of the RAINBOW approach to perform an interactive conceptual analysis, based on the reverse engineering of prototypical user-drawn form-based interfaces, especially for environments where forms are a privileged way to exchange information and stakeholders are familiar with form-based (computer) interaction and the application domain. In order to overcome the observed limitations

of related approaches and transparently produce a conceptual schema of the application domain that includes hierarchies, constraints and dependencies, we formalised the approach into a semi-automatic seven-step process specialising and integrating standard techniques to help acquire data specifications from existing artefacts.

The *Represent* step first focused on the drawing and specification of a set of simple form-based interfaces that would enable end-users to perform usual tasks of their application domain. For this purpose, we notably explained how the richness and inherent complexity of existing UIDLs led us to define RAINBOW's simplified form model, based on the most common form widgets that are `forms`, `fieldsets`, `tables`, `inputs`, `selections` and `buttons`. We then exposed how to manage the drawing step, by preparing and planning the project, training the end-users, and finally providing them with a tool-support consistent with our form model.

We then explained how to translate the produced set of form-based interfaces into a corresponding set of data models through the *Adapt* step. For this purpose, we presented then formalised intuitive mapping rules to support this extraction and obtain simple but semantically equivalent data structures of the GER model.

The *Investigate* step subsequently addressed the analysis of these data models to highlight semantic and structural similarities, which were formalised based on the definition of orthographic and ontological similarities, as well as the use of patterns. We also presented how to process the arbitrated similarities in order to produce a pre-integrated schema with unified terminology and structures, and containing the materialisation of the relationships between the concepts conveyed by the form containers.

The *Nurture* step then addressed the elicitation of technical, existence and unique constraints, as well as functional dependencies and the parallel acquisition of data samples. We formalised these notions and presented the principles of an interactive process, inspired by Armstrong relations, in order to suggest them, collect them and consequently reflect them on the pre-integrated schema.

The *Bind* step afterwards handled and formalised the arbitration and processing of the previously defined constraints and dependencies, as well as the relationships specified between entity types, in order to produce an integrated conceptual schema representing the application domain for which the form-based interfaces were originally drawn.

Finally, the *Objectify* step addressed the generation of a lightweight prototypical data manager application from this integrated conceptual schema, and the *Wander* step exposed how to submit this prototype to the end-users in order to transparently refine and ultimately validate the integrated con-

ceptual schema. We then presented the prototypical RAINBOW Toolkit that was developed in order to support and experiment the approach and provide a sequential access and support to the five first crucial steps.

As could be expected, we then addressed the intrinsically complex validation of the RAINBOW approach. In particular, we focused on its effectiveness (i.e. its ability to help end-users and analysts to communicate static data requirements to each other), and the quality of the conceptual schemas produced using it. The issues raised by these quintessential questions are not easy to experiment, measure and validate, especially given the immanent difficulty of evaluating methodologies for the development of large systems, which primarily requires to spread the experimentations over time. Therefore, we defined an experimentation canvas that we applied to two preliminary studies, in order to get a first insight on the validation method and the implementation of the RAINBOW approach.

We subsequently exposed our validation protocol based on the Participant-Observer principles to monitor the use of the RAINBOW toolkit and approach, and the Brainstorming/Focus group principles to analyse the resulting conceptual schemas. We consequently proposed to structure each experimentation into a preparation phase, an execution phase and finally a review phase, then detailed how the participating end-users, analysts and observers were involved in each of these phases. We then introduced the two case studies, which were each rich and relevant in their own way, and described how each pair of end-user and analyst managed to jointly design the conceptual schema of their application project using the RAINBOW methodology and toolkit, while the observers took notes about the efficiency of the process. We followed with the resulting discussions on the quality of the schemas produced using the approach and tool support for each study.

The analysis of these preliminary experimentations led us to conclude that the experimentation canvas proved to be valid and relevant. Besides, the RAINBOW approach and tool support did effectively help end-users and analysts to communicate static data requirements to each other and that the quality of the produced conceptual schemas was good with respect to the given case studies. This encouraging preliminary validation process also highlighted several sensible phenomenons that will need to be monitored on a larger scale experimentation, such as the drawing behaviour of the participants.

Finally, we retrospected on the RAINBOW approach to assess its specificities and merits, as well as its flaws and possible future works. We particularly recalled some of the challenges that were overcame in order to intimately involve end-users in the data elicitation processes, and underlined the importance of the analyst in that matter. We also explained how principles of Reverse En-

gineering were applied on controlled artefacts and through monitored processes to elicit requirements for the purpose of Forward Engineering. We discussed the modularity and transformation-based reliability of the overall process, as well as the diversity of potential users, before arguing that the output of the approach was a rich and relevant part of a valid Software Requirement Specification that could be used in conjunction with other elicitation methodologies.

Among the possible improvements, we recalled that the last two steps of the RAINBOW approach could also be formalised, though they are relatively straightforward. We also mentioned that the approach could take advantage of dynamic aspects of form-based interfaces, reusability mechanisms for the elaboration of the interfaces, refinements of the terminological and structural analysis, as well as the expansion of constraints elicitation and the study of data samples. The tool support could also be improved by notably working on its ergonomy and usability, which could progressively turn it into a true CASE tool. Finally, we drew the main lines of an improved experimentation canvas for a wider experimentation endeavour over time that could give us a better understanding of the real impact of the RAINBOW approach. In order to get the most of these futures experiments, we advocated to carefully pave the way for the procedure, and set guidelines for the execution as well as the reviewing and comparison of the approach with similar existing approaches.

In the end, we can finally conclude that the RAINBOW approach qualifies as an original and realistic contribution to elicit static database requirements in the context of Software Engineering. As expected, the expressiveness of form-based interfaces and prototypes, combined with the specialisation and integration of standard technique to help acquire and validate specifications from existing artefacts, enabled to use form-based interfaces as a two-way communication channel to communicate static data requirements between end-users and analysts. This approach can evidently be extended and optimised, but nevertheless, it overcomes the main concerns raised by similar researches, while being interoperable with other approaches and extensible for further analysis and elicitation processes. Besides, the experimentation results of the preliminary studies comforts us in believing that this approach is viable, worthy, and deserves to be improved and tested over time, by continuously looking for better ways to involve stakeholders in an efficient and satisfying fashion.

Part V

Bibliography

Give back to Caesar what is Caesar's and to God what is God's.
Matthew, XXII, 21

References

Ali, M. F., Pérez-Quiñones, M. A., Abrams, M., and Shell, E. (2002). Building multi-platform user interfaces with uiml. In Kolski and Vanderdonckt [2002], pages 255–266. [cited at p. 55]

Andriole, S. J. (1994). Fast, cheap requirements: Prototype, or else! *IEEE Software*, 11(2):85–87. [cited at p. 13]

Armstrong, W. W. (1974). Dependency structures of data base relationships. In *IFIP Congress*, pages 580–583. [cited at p. 46]

Asai, T., Arimura, H., Uno, T., ichi Nakano, S., and Satoh, K. (2003). Efficient tree mining using reverse search. In *International Symposium on Information Science and Electrical Engineering 2003 (ISEE 2003), Kyushu University*, pages 401–404. [cited at p. 43]

Astrova, I. and Stantic, B. (2005). An html-form-driven approach to reverse engineering of relational databases to ontologies. In *Proceedings of IASTED International Conference on Databases and Applications*, pages 246–251. [cited at p. 33]

Baixeries, J. (2004). A formal concept analysis framework to mine functional dependencies. In *Proceeding of Mathematical Methods for Learning 2004 : Advances in data mining and knowledge discovery.* [cited at p. 48]

Batini, C., Ceri, S., and Navathe, S. B. (1992). *Conceptual database design: an Entity-relationship approach.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA. [cited at p. 14, 15, 25, 197]

Batini, C., Demo, G. B., and Leva, A. D. (1984). A methodology for conceptual design of office data bases. *Information Systems*, 9(3/4):251–263. [cited at p. 32]

Batini, C., Lenzerini, M., and Navathe, S. B. (1986). A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364. [cited at p. 49]

Bødker, S., Grønbaek, K., and Kyng, M. (1993). Cooperative design: Techniques and experience from the scandinavian scene. In Schuler, D. and Namioka, A., editors, *Participatory design: Principles and practices*. Hillsdale, New Jersey: Lawrence Erlbaum Associates. [cited at p. 57]

Beyer, H. and Holtzblatt, K. (1998). *Contextual design: defining customer-centered systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. [cited at p. 11]

Briand, L. C., Morasca, S., and Basili, V. R. (1996). Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86. [cited at p. 195]

Brogneaux, A.-F., Ramdoyal, R., Vilz, J., and Hainaut, J.-L. (2005a). Deriving user-requirements from human-computer interfaces. In *Proceedings of 23rd IASTED International Conference, Innsbruck, Austria*, pages 77–82. [cited at p. 59]

Brogneaux, A.-F., Ramdoyal, R., Vilz, J., and Hainaut, J.-L. (2005b). Deriving user-requirements from human-computer interfaces. In *TC13 Workshop on Human-Computer Interaction (Design and Visualisation), Namur, Belgium*. [cited at p. 59]

Brown, J. S. and Duguid, P. (2000). *The Social Life of Information*. Harvard Business School Press, Boston, MA, USA. [cited at p. 27]

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. (2003). A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308. [cited at p. 257]

Carnegie Mellon University (2006). The SecondString project, the open-source java-based package of approximate string-matching techniques. http://secondstring.sourceforge.net. [cited at p. 179]

Chapman, S. (2007). SimMetrics, the open source extensible library of similarity or distance metrics. http://simmetrics.sourceforge.net. [cited at p. 179]

Chehreghani, M. H., Rahgozar, M., Lucas, C., and Chehreghani, M. H. (2007). Mining maximal embedded unordered tree patterns. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2007,Honolulu, Hawaii, USA*, pages 437–443. [cited at p. 43]

Cherfi, S. S.-S., Akoka, J., and Comyn-Wattiau, I. (2002). Conceptual modeling quality - from eer to uml schemas evaluation. In *ER '02: Proceedings of the 21st International Conference on Conceptual Modeling*, pages 414–428, London, UK. Springer-Verlag. [cited at p. 195]

Chi, Y., Muntz, R. R., Nijssen, S., and Kok, J. N. (2005). Frequent subtree mining - an overview. *Fundamenta Informatica*, 66(1-2):161–198. [cited at p. 43]

Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17. [cited at p. 16, 197]

Choobineh, J., Mannino, M. V., and Tseng, V. P. (1992). A form-based approach for database analysis and design. *Communications of the ACM*, Vol. 35, Nř2:108–120. [cited at p. 27, 33, 43, 81, 252]

Coad, P. (1992). Object-oriented patterns. *Communications of the ACM*, 35(9):152–159. [cited at p. 123]

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387. [cited at p. 45]

Codd, E. F. (1971a). Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909. [cited at p. 195]

Codd, E. F. (1971b). Normalized data structure: A brief tutorial. In Codd, E. F. and Dean, A. L., editors, *SIGFIDET Workshop*, pages 1–17. ACM. [cited at p. 195]

Cohen, W. W., Ravikumar, P., and Fienberg, S. E. (2003). A comparison of string distance metrics for name-matching tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), Acapulco, Mexico*, pages 73–78. [cited at p. 40]

Connell, J. and Shafer, L. I. (1995). *Object-oriented rapid prototyping*. Yourdon Press, Upper Saddle River, NJ, USA. [cited at p. 27]

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702. [cited at p. 53]

Correia, J. H. (2002). Relational scaling and databases. In *Proceedings of the 10th International Conference on Conceptual Structures (ICCS 2002), Borovets, Bulgaria, July 15-19, 2002*, pages 62–76. [cited at p. 48]

Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. (1996). Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132. [cited at p. 195]

Davis, A. M. and Zowghi, D. (2006). Good requirements practices are neither necessary nor sufficient. *Requirements Engineering*, 11(1):1–3. [cited at p. 7]

DB-MAIN (2010). The DB-MAIN CASE Tool. http://www.db-main.be. [cited at p. 51, 175]

Dix, A., Finley, J., Abowd, G., and Beale, R. (1998). *Human-computer interaction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. [cited at p. 8]

Embley, D. W. (1989). NFQL: The natural forms query language. *ACM Transactions on Database Systems*, 14(2):168–211. [cited at p. 173]

Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. MIT Press. [cited at p. 108, 179]

Finlayson, M. A. (2009). The MIT Java Wordnet Interface (JWI). http://projects.csail.mit.edu/jwi/. [cited at p. 179]

Fischer, G. (2002). Beyond "couch potatoes": From consumers to designers and active contributors. *First Monday*, 7(12). [cited at p. 12, 27]

Flory, A. (1982). *Bases de données : conception et réalisation*. ECONOMICA, Paris. [cited at p. 49]

Gemino, A. and Wand, Y. (2005). Complexity and clarity in conceptual modeling: Comparison of mandatory and optional properties. *Data & Knowledge Engineering*, 55(3):301–326. [cited at p. 195]

Genero, M., Jiménez, L., and Piattini, M. (2000). Measuring the quality of entity relationship diagrams. In *ER'00: Proceedings of the 19th international conference on Conceptual modeling*, pages 513–526, Berlin, Heidelberg. Springer-Verlag. [cited at p. 195]

Goguen, J. A. and Linde, C. (1993). Techniques for requirements elimination. In *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 152–164, Los Alamitos, California. IEEE CS Press. [cited at p. 9]

Gomaa, H. (1983). The impact of rapid prototyping on specifying user requirements. *SIGSOFT Software Engineering Notes*, 8(2):17–27. [cited at p. 26]

Gomaa, H. and Scott, D. B. (1981). Prototyping as a tool in the specification of user requirements. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*, pages 333–342. IEEE Press. [cited at p. 197]

Grønbæk, K., Kyng, M., and Mogensen, P. (1997). Toward a cooperative experimental system development approach. In Kyng, M. and Mathiassen, L., editors, *Computers and design in context*, pages 201–238. MIT Press, Cambridge, MA, USA. [cited at p. 58]

Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies*, 43(5-6):907 – 928. [cited at p. 40]

Habra, N., Abran, A., Lopez, M., and Sellami, A. (2008). A framework for the design and verification of software measurement methods. *Journal of Systems and Software*, 81(5):633–648. [cited at p. 195]

Hainaut, J.-L. (1989). A generic entity-relationship model. In *Proceedings of the IFIP WG 8.1 Conference on Information System Concepts: an in-depth analysis*, pages 109–138. North-Holland. [cited at p. 17]

Hainaut, J.-L. (1996). Specification preservation in schema transformations - application to semantics and statistics. *Data & Knowledge Engineering*, 19(2):99–134. [cited at p. 25]

Hainaut, J.-L. (2002). *Introduction to Database Reverse Engineering, 3rd Edition.* LIBD Publish., Namur. http://www.info.fundp.ac.be/dbm/publication/2002/DBRE-2002.pdf. [cited at p. 16, 28, 254]

Hainaut, J.-L. (2005). Transformation-based database engineering. In van Bommel, P., editor, *Transformation of Knowledge, Information and Data: Theory and Applications*, chapter 1. IDEA Group. [cited at p. 81]

Hainaut, J.-L. (2006). The transformational approach to database engineering. In Lämmel, R., Saraiva, J., and Visser, J., editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*, pages 95–143. Springer. [cited at p. 14, 15, 22, 24, 28, 197]

Hainaut, J.-L. (2009). *Bases de données - Concepts, utilisation et développement, 3rd Edition.* Dunod. [cited at p. 50]

Hainaut, J.-L., Henrard, J., Hick, J.-M., Roland, D., and Englebert, V. (1996). Database design recovery. In *Proceedings of International Conference on Advances Information System Engineering (CAiSE)*, volume 1080 of *LNCS*, pages 272–300. Springer. [cited at p. 26]

Hall, P. A. V. (1992). *Software Reuse and Reverse Engineering in Practice.* Chapman & Hall, Ltd., London, UK, UK. [cited at p. 16, 197]

Hersh, W., Price, S., and Donohoe, L. (2000). Assessing thesaurus-based query expansion using the umls metathesaurus. In *Proceedings of the 2000 American Medical Informatics Association (AMIA) Symposium*, pages 344–348. [cited at p. 41]

Hick, J.-M. and Hainaut, J.-L. (2006). Database application evolution: A transformational approach. *Data & Knowledge Engineering*, 59:534–558. [cited at p. 26]

Hoxmeier, J. A. (1998). Typology of database quality factors. *Software Quality Control*, 7(3/4):179–193. [cited at p. 195]

Huhtala, Y., Kärkkäinen, J., Porkka, P., and Toivonen, H. (1999). TANE: An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal*, 42(2):100–111. [cited at p. 48]

IEEE (1998). IEEE recommended practice for software requirements specifications. Technical report, IEEE. [cited at p. 10]

Illich, I. (1973). *Tools for Conviviality.* Harper & Row Publishers, New York. [cited at p. 12, 27]

ISO/IEC (2001). *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC. [cited at p. 194]

Jacobs, J. (1982). Finding words that sound alike. the soundex algorithm. *Byte 7*, pages 473–474. [cited at p. 40]

Java (2010). The Java official website. http://www.java.com. [cited at p. 175]

Jiménez, A., Berzal, F., and Cubero, J. C. (2008). Mining induced and embedded subtrees in ordered, unordered, and partially-ordered trees. In *Proceedings of Foundations of Intelligent Systems, 17th International Symposium, ISMIS 2008, Toronto, Canada*, pages 111–120. [cited at p. 43]

Kensing, F. and Blomberg, J. (1998). Participatory design: Issues and concerns. *Computer Supported Cooperative Work*, 7(3/4):167–185. [cited at p. 57]

Kesh, S. (1995). Evaluating the quality of entity relationship models. *Information and Software Technology*, 37(12):681 – 689. [cited at p. 195]

Kolski, C. and Vanderdonckt, J., editors (2002). *Computer-Aided Design of User Interfaces III, Proceedings of the Fourth International Conference on Computer-Aided Design of User Interfaces, May, 15-17, 2002, Valenciennes, France.* Kluwer. [cited at p. 279, 287]

Kösters, G., Six, H.-W., and Voss, J. (1996). Combined analysis of user interface and domain requirements. In *ICRE '96: Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96)*, page 199, Washington, DC, USA. IEEE Computer Society. [cited at p. 34, 252]

Krogstie, J. (1998). Integrating the understanding of quality in requirements specification and conceptual modeling. *SIGSOFT Software Engineering Notes*, 23(1):86–91. [cited at p. 195]

Lantz, K. E. (1986). *The prototyping methodology.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA. [cited at p. 26, 197]

Lee, H. and Yoo, C. (2000). A form-driven object-oriented reverse engineering methodology. *Information Systems*, Vol. 25, No. 3. [cited at p. 33]

Limbourg, Q. and Vanderdonckt, J. (2004). Usixml: A user interface description language supporting multiple levels of independence. In *Proceedings of Workshops in connection with the 4th International Conference on Web Engineering (ICWE 2004), Munich, Germany*, pages 325–338. [cited at p. 56]

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and López-Jaquero, V. (2004). Usixml: A language supporting multi-path development of user interfaces. In Bastide, R., Palanque, P. A., and Roth, J., editors, *EHCI/DS-VIS*, volume 3425 of *Lecture Notes in Computer Science*, pages 200–220. Springer. [cited at p. 56]

Lindland, O. I., Sindre, G., and Sølvberg, A. (1994). Understanding quality in conceptual modeling. *IEEE Software*, 11(2):42–49. [cited at p. 195]

LingPipe (2010). The LingPipe tool kit for processing text using computational linguistics. http://alias-i.com/lingpipe/. [cited at p. 179]

Lopes, S., Petit, J.-M., and Lakhal, L. (2000). Efficient discovery of functional dependencies and armstrong relations. In *Proceedings of Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany*, pages 350–364. [cited at p. 48]

Lopes, S., Petit, J.-M., and Lakhal, L. (2002). Functional and approximate dependency mining: database and fca points of view. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 14(2-3):93–114. [cited at p. 48]

Luyten, K., Abrams, M., Vanderdonckt, J., and Limbourg, Q. (2004). Developing user interfaces with xml: Advances on user interface description languages. In *Proceedings of the Satellite Workshop of Advanced Visual Interfaces, Gallipoli, Italy*. [cited at p. 54]

Maes, A. and Poels, G. (2006). Evaluating quality of conceptual models based on user perceptions. In *Proceedings of ER 2006, 25th International Conference on Conceptual Modeling, Tucson, AZ, USA, November 6-9, 2006*, pages 54–67. [cited at p. 195]

Mayo, E. (1933). *The Human Problems of an Industrial Civilization*. Mac Millan, New York. [cited at p. 13, 192]

McConnell, S. (2000). From the editor - the best influences on software engineering. *IEEE Software*, 17(1). [cited at p. 7]

Mehandjiev, N., Layzell, P., Brereton, P., Lewis, G., Mannion, M., and cois Coallier, F. (2002). Thirteen knights and the seven-headed dragon: an interdisciplinary software engineering framework. In *STEP '02: Proceedings of the 10th International Workshop on Software Technology and Engineering Practice*, page 46, Washington, DC, USA. IEEE Computer Society. [cited at p. 6]

Mfourga, N. (1997). Extracting entity-relationship schemas from relational databases: A form-driven approach. *Reverse Engineering, Working Conference on*, 0:184. [cited at p. 32]

Mogensen, P. (1992). Towards a provotyping approach in systems development. *Scandinavian Journal of Information Systems*, 4:31–53. [cited at p. 58]

Moody, D. (2006). What makes a good diagram? improving the cognitive effectiveness of diagrams in is development. In Knapp and Magyar, editors, *Intl Conf on Information Systems Development*. Springer. [cited at p. 195]

Moody, D. L. and Shanks, G. G. (2003). Improving the quality of data models: empirical validation of a quality management framework. *Information Systems*, 28(6):619–650. [cited at p. 195]

Moody, D. L., Sindre, G., Brasethvik, T., and Sølvberg, A. (2003). Evaluating the quality of information models: empirical testing of a conceptual model quality framework. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 295–305, Washington, DC, USA. IEEE Computer Society. [cited at p. 195]

Mori, G., Paternò, F., and Santoro, C. (2002). Ctte: support for developing and analyzing task models for interactive system design. *IEEE Transactions on Software Engineering*, 28(8):797–813. [cited at p. 56]

Muller, M. J., Wildman, D. M., and White, E. A. (1993). Taxonomy of pd practices: A brief practitioner's guide. *Communications of the ACM*, 36(6):26–28. [cited at p. 58]

Naur, P., Randell, B., and Buxton, J. (1976). *Software Engineering: Concepts and Techniques*. Petrocelli/Carter, New York, USA. [cited at p. 6]

Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88. [cited at p. 40]

Nielsen, J. (1986). A virtual protocol model for computer-human interaction. *International Journal of Man-Machine Studies*, 24(3):301–312. [cited at p. 91]

Novelli, N. and Cicchetti, R. (2001). FUN: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of Database Theory - ICDT 2001, 8th International Conference, London, UK*, pages 189–203. [cited at p. 48]

Nuseibeh, B. and Easterbrook, S. (2000). Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA. ACM Press. [cited at p. 7, 9]

Object Management Group (OMG) (2007). Introduction to omg's unified modeling language. http://www.omg.org/gettingstarted/what_is_uml.htm. [cited at p. 10, 78]

Osmundson, J. S., Michael, J. B., Machniak, M. J., and Grossman, M. A. (2003). Quality management metrics for software development. *Information and Management*, 40(8):799–812. [cited at p. 6]

Pastor, O., Gómez, J., Insfrán, E., and Pelechano, V. (2001). The oo-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. *Information Systems*, 26(7):507–534. [cited at p. 257]

Pastor, O., Hayes, F., and Bear, S. (1992). Oasis: An object-oriented specification language. In *Proceedings of the Advanced Information Systems Engineering, CAiSE'92, Manchester, UK*, pages 348–363. [cited at p. 257]

Pastor, O. and Insfrán, E. (2003). Oo-method, the methodological support for oliva nova model execution system. White paper, CARE Technologies S.A. http://www.care-t.com/technology/whitepapers.asp. [cited at p. 257]

Paternò, F. and Santoro, C. (2002). One model, many interfaces. In Kolski and Vanderdonckt [2002], pages 143–154. [cited at p. 56]

Pomberger, G., Bischofberger, W. R., Kolb, D., Pree, W., and Schlemm, H. (1991). Prototyping-oriented software development - concepts and tools. *Structured Programming*, 12(1):43–60. [cited at p. 27]

President's Information Technology Advisory Committee (PITAC) (1999). Information technology research: Investing in our future, report to the president. http://www.nitrd.gov/pitac/report/. [cited at p. 6]

Pressman, R. S. (2000). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education. [cited at p. 12]

Priss, U. (2005). Establishing connections between formal concept analysis and relational databases. In *Common Semantics for Sharing Knowledge: Contributions to ICCS 2005*, pages 132–145. [cited at p. 48]

Puerta, A. R. and Eisenstein, J. (2002). Ximl: a common representation for interaction data. In *IUI*, pages 216–217. [cited at p. 55]

Qt (2010). The QT official website. http://qt.nokia.com. [cited at p. 175]

Rahm, E. and Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350. [cited at p. 39]

Ram, S. (1995). Deriving functional dependencies from the entity-relationship model. *Communications of the ACM*, 38(9):95–107. [cited at p. 47]

Ramdoyal, R. (2010). Reverse engineering user-drawn form-based interfaces for interactive database conceptual analysis. In *Proceedings of CAiSE Doctoral Consortium 2010, Hammamet, Tunisia*. [cited at p. 59]

Ramdoyal, R., Brogneaux, A.-F., Vilz, J., and Hainaut, J.-L. (2007). Recherche de recouvrements dans une collection de schémas de base de données. In *Proceedings of the DECOR Workshop, EGC 2007, Namur, Belgium*. [cited at p. 59]

Ramdoyal, R., Cleve, A., Brogneaux, A.-F., and Hainaut, J.-L. (2009). Rétro-ingénierie dŠinterfaces utilisateur pour lŠanalyse conceptuelle de bases de données. In *Proceedings of the 25èmes Journées en Bases de Données Avancées (BDA 2009), Namur, Belgium.* [cited at p. 59]

Ramdoyal, R., Cleve, A., and Hainaut, J.-L. (2010). Reverse engineering user interfaces for interactive database conceptual analysis. In *Proceedings of CAiSE 2010, Hammamet, Tunisia*, volume 6051 of *LNCS*, pages 332–347. [cited at p. 59]

Rancz, K. T. J. and Varga, V. (2008). A method for mining functional dependencies in relational database design using fca. *Studia Universitatis "Babes-Bolyait't' Cluj-Napoca, Informatica*, Volume LIII(1):17Ű28. [cited at p. 48]

Rancz, K. T. J., Varga, V., and Puskas, J. (2008). A software tool for data analysis based on formal concept analysis. *Studia Universitatis "Babes-Bolyait't' Cluj-Napoca, Informatica*, Volume LIII(2):67–78. [cited at p. 49]

Ravid, A. and Berry, D. M. (2000). A method for extracting and stating software requirements that a user interface prototype contains. *Requirements Engineering*, 5(4):225–241. [cited at p. 28]

Robbins-Gioia LLC (2002). ERP survey. http://www.robbinsgioia.com/news%5Fevents/012802%5Ferp.aspx. [cited at p. 12]

Rode, J., Bhardwaj, Y., Pérez-Quiñones, M. A., Rosson, M. B., and Howarth, J. (2005). As easy as "click": End-user web engineering. In Lowe, D. and Gaedke, M., editors, *Proceedings of the 5th International Conference on Web Engineering, ICWE 2005, Sydney, Australia, July 27-29, 2005*, volume 3579 of *Lecture Notes in Computer Science*, pages 478–488. Springer. [cited at p. 35, 252]

Rollinson, S. R. and Roberts, S. A. (1998). Formalizing the informational content of database user interfaces. In *ER'98: Proc. of the 17th International Conference on Conceptual Modeling*, pages 65–77. Springer-Verlag. [cited at p. 28, 35, 81, 252]

Sanders, E. B.-N. (2002). From user-centered to participatory design approaches. In Frascara, J., editor, *Design and the Social Sciences*. Taylor & Francis Books Limited. [cited at p. 57]

Schewe, K. D. and Thalheim, B. (2005). Conceptual modelling of web information systems. *Data & Knowledge Engineering*, 54(2):147–188. [cited at p. 14]

Schmidt, D. C. (2006). Model-driven engineering. *IEEE Computer*, 39(2):25–31. [cited at p. 256]

Schneider, K. (1996). Prototypes as assets, not toys: why and how to extract knowledge from prototypes. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 522–531, Washington, DC, USA. IEEE Computer Society. [cited at p. 27]

Schuler, D. and Namioka, A., editors (1993). *Participatory Design: Principles and Practices*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA. [cited at p. 11, 197]

Sharp, H., Finkelstein, A., and Galal, G. (1999). Stakeholder identification in the requirements engineering process. In *DEXA '99: Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, page 387, Washington, DC, USA. IEEE Computer Society. [cited at p. 8]

Shoval, P. and Shiran, S. (1997). Entity-relationship and object-oriented data modeling-an experimental comparison of design quality. *Data & Knowledge Engineering*, 21(3):297–315. [cited at p. 15]

Singer, J., Sim, S. E., and Lethbridge, T. C. (2008). Software engineering data collection for field studies. In Shull, F., Singer, J., and Sjøberg, D. I., editors, *Guide to Advanced Empirical Software Engineering*, pages 9–34. Springer. [cited at p. 192]

Sommerville, I. and Kotonya, G. (1998). *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc., New York, NY, USA. [cited at p. 9]

Souchon, N. and Vanderdonckt, J. (2003). A review of xml-compliant user interface description languages. In Jorge, J. A., Nunes, N. J., and e Cunha, J. F., editors, *DSV-IS*, volume 2844 of *Lecture Notes in Computer Science*, pages 377–391. Springer. [cited at p. 56]

Spaccapietra, S., Parent, C., and Dupont, Y. (1992). Model independent assertions for integration of heterogeneous schemas. *The VLDB Journal*, 1(1):81–126. [cited at p. 50]

Spell, B. (2009). Java API for WordNet Searching (JAWS). http://lyle.smu.edu/~tspell/jaws. [cited at p. 179]

Standish Group International Inc. (1995). Chaos report. http://www.standishgroup.com/chaos. [cited at p. 6]

Standish Group International Inc. (1999). Chaos : A recipe for success. http://www.standishgroup.com/chaos. [cited at p. 7]

Standish Group International Inc. (2001). Extreme chaos. http://www.standishgroup.com/chaos. [cited at p. 7]

Stroulia, E., El-Ramly, M., Kong, L., Sorenson, P. G., and Matichuk, B. (1999). Reverse engineering legacy interfaces: An interaction-driven approach. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99), Atlanta, USA*, pages 292–302. [cited at p. 32]

Termier, A., Rousset, M.-C., and Sebag, M. (2002). Treefinder: a first step towards xml data mining. In *Second IEEE International Conference on Data Mining (ICDM'02)*, pages 450–457. [cited at p. 43]

Terwilliger, J. F., Delcambre, L. M. L., and Logan, J. (2006). The user interface is the conceptual model. In *Proceedings of 25th International Conference on Conceptual Modeling (ER'06)*, volume 4215 of *LNCS*, pages 424–436. Springer. [cited at p. 27, 35, 173, 252]

Trigg, R. H., Bødker, S., and Grønbæk, K. (1991). Open-ended interaction in cooperative prototyping a video-based analysis. *Scandinavian Journal of Information Systems*, 3:63–86. [cited at p. 58]

UsiXML Consortium (2007). Usixml v1.8 reference manual. http://www.usixml.org. [cited at p. 68]

Vilz, J., Brogneaux, A.-F., Ramdoyal, R., Englebert, V., and Hainaut, J.-L. (2006). Data conceptualisation for web-based data-centred application design. In *Proceedings of the Advanced Information Systems Engineering, 18th International Conference, CAiSE 2006, Luxembourg*, LNCS, pages 205–219. [cited at p. 43, 59, 63]

Vosburgh, J., Curtis, B., Wolverton, R., Albert, B., Malec, H., Hoben, S., and Liu, Y. (1984). Productivity factors and programming environments. In *ICSE*, pages 143–152. [cited at p. 11]

Vries, H. D., Verheul, H., and Willemse, H. (2003). Stakeholder identification in it standardisation processes. In *MIS Quarterly Special Issue Workshop on Standard Making: A Critical Research Frontier for Information Systems*, pages 92–107. [cited at p. 8]

Wahler, M. (2008). *Using Patterns to Develop Consistent Design Constraints*. PhD thesis, ETH Zurich, Switzerland. [cited at p. 195]

Walenz, B. and Didion, J. (2008). The Java WordNet Library (JWNL). http://jwordnet.sourceforge.net. [cited at p. 179]

Wille, R. (2005). Formal concept analysis as mathematical theory of concepts and concept hierarchies. In *Formal Concept Analysis, Foundations and Applications*, volume 3626 of *Lecture Notes in Computer Science*, pages 1–33. Springer. [cited at p. 43]

Wilson, W. M., Rosenberg, L. H., and Hyatt, L. E. (1997). Automated analysis of requirement specifications. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 161–171, New York, NY, USA. ACM. [cited at p. 12]

Winkler, W. E. (1990). String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research Methods, American Statistical Association*, pages 472–477. [cited at p. 40, 104, 179]

World Wide Web Consortium (W3C) (2010). The Extensible Markup Language (XML). http://www.w3.org/XML. [cited at p. 55, 180]

Wyss, C. M., Giannella, C., and Robertson, E. L. (2001). Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *Proceedings of Data Warehousing and Knowledge Discovery, Third International Conference, DaWaK 2001, Munich, Germany*, pages 101–110. [cited at p. 48]

Yang, F., Gupta, N., Botev, C., Churchill, E. F., Levchenko, G., and Shanmugasundaram, J. (2008). Wysiwyg development of data driven web applications. *Proceedings of the VLDB Endowment*, 1(1):163–175. [cited at p. 36, 252]

Yao, H. and Hamilton, H. J. (2008). Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2):197–219. [cited at p. 48]

Zaki, M. J. (2005). Efficiently mining frequent embedded unordered trees. *Fundamenta Informatica*, 66(1-2):33–52. [cited at p. 43]

Part VI

Appendices

Among the appendices, Chapters A and B respectively provide explanations for the conventions used in the algorithms and the schemas presented in this dissertation, while Chapter C supplies information to obtain additional materials related to this doctoral research.

Appendix A

Algorithmic conventions

In this chapter, we present the conventions used to express the algorithms of this dissertation. These high level algorithms rely on basic conditional (`if/then/else`) and iterative (`for all/do` and `while/do`) structures, assignments (`x` ← `y`), user input (`Ask`) and calls to existing or predefined methods or algorithms, as illustrated by Algorithm A.1.

Algorithm A.1 MyAlgorithm : A sample algorithm to illustrate the conventions

Require: the necessary preconditions
Ensure: the resulting postconditions

```
 1: procedure MyAlgorithm(p1, p2, p3, ...)  ▷ The algorithm's name and parameters
 2:     x ← y  ▷ Assign the value (of) y to variable x
 3:     Ask> define: x  ▷ Ask the user to define x
 4:     Ask> choose/redefine: x  ▷ Ask the user to choose or redefine x
 5:     AnotherAlgorithm(p'1, p'2, p'3, ...)  ▷ Call another algorithm with the appropriate parameters
 6:     if condition1 then  ▷ A conditional structure
 7:         ...  ▷ Do something
 8:     else if condition2 then
 9:         ...  ▷ Do something else
10:     else
11:         ...  ▷ Proceed with the default instructions
12:     end if
13:     for all x verifying a condition do  ▷ An iterative structure
14:         ...  ▷ Do something
15:     end for
16:     while condition do  ▷ Another iterative structure
17:         ...  ▷ Do something
18:     end while
19:      ▷ And by the way, this is a comment.
20: end procedure
```

Appendix B

Schemas representation conventions

In this chapter, we present the conventions used to graphically represent the schemas of this dissertation. Recall, as exposed in Section 6.3, that in the scope of this research, we work with a sub-model of the GER model which is restricted to "flat" entity types (i.e. entity types having only atomic attributes), binary relationship types (i.e. relationship types having exactly two roles) and IS-A hierarchies. This sub-model encompasses part of concepts illustrated in the following figures.

Figure B.1 illustrates basic GER concepts, which include:

- schemas;
- entity types;
- attributes, atomic or compound, optional or mandatory;
- binary relationship types;
- roles;
- cyclic relationship types;
- primary and secondary identifiers, either attribute-based or hybrid.

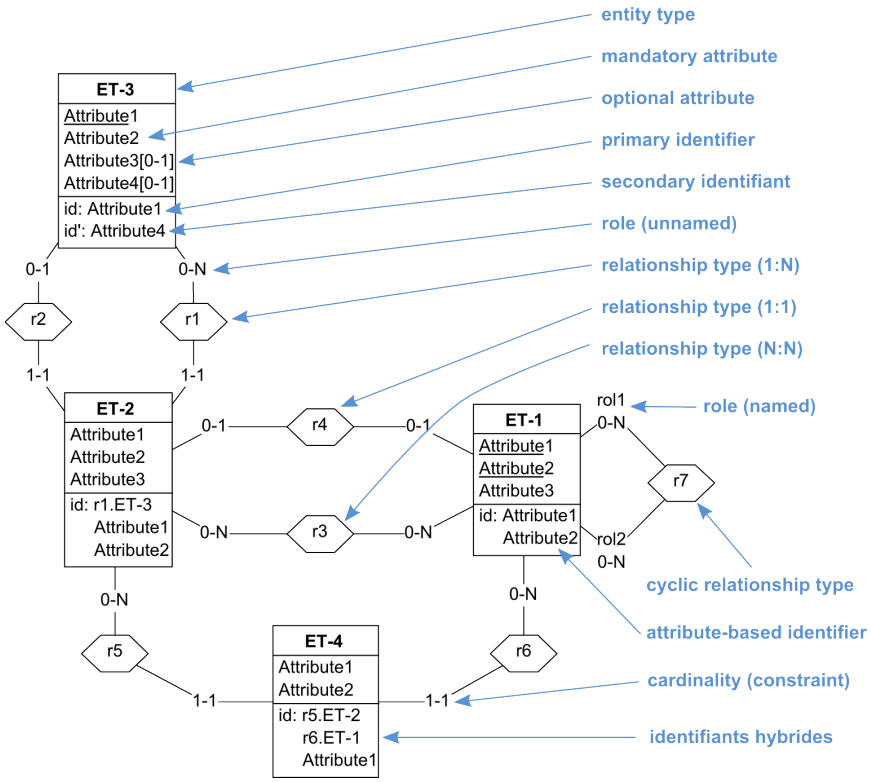


Figure B.1: Basic GER concepts.

Figure B.2 illustrates IS-A relations, which involve entity types with unique or multiple supertypes, as well as disjunction, totality and partition constraints.

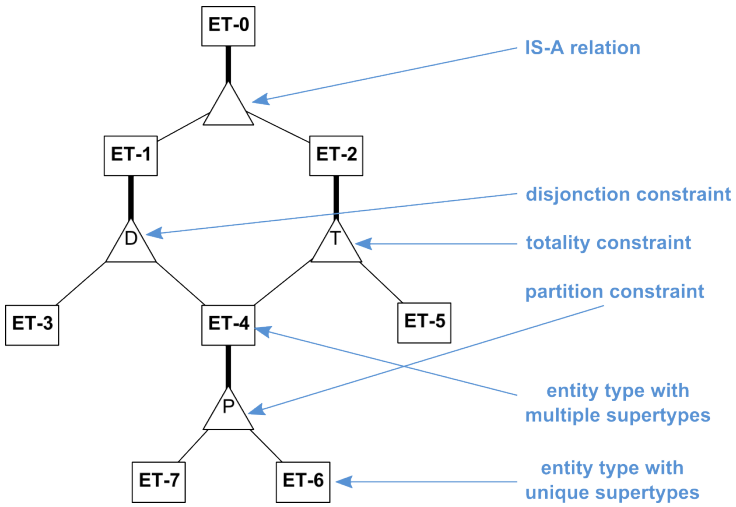


Figure B.2: IS-A relations.

Figure B.3 illustrates advanced GER constructs, among which:

- stereotypes;
- compound and/or multivalued attributes;
- user defined domains (UDD);
- multivalued identifiers for entity types;
- attribute identifiers;
- procedural units.

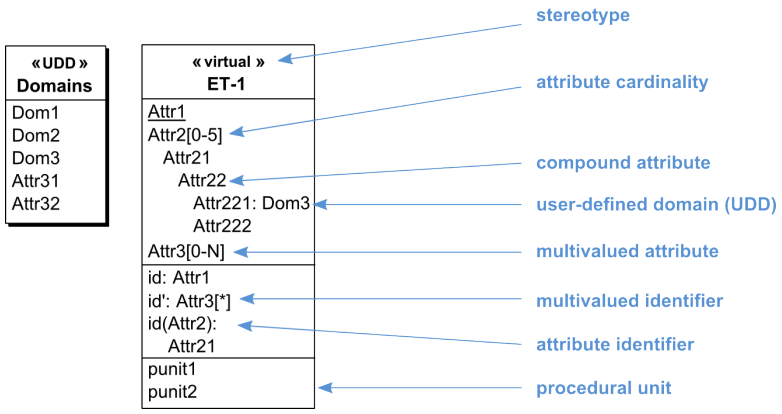


Figure B.3: Stereotypes, attributes, domains and procedural units.

Figure B.4 illustrates different types of constraints, among which we can notably mention existence constraints (coexistence, at least one, at most one, exactly one).

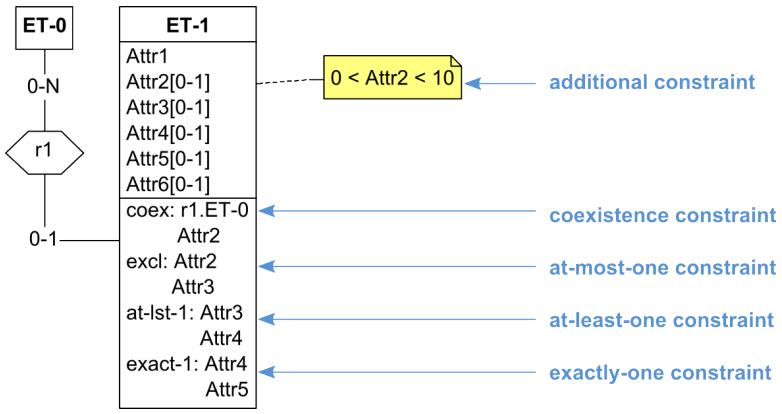


Figure B.4: Different types of constraints

Appendix C

Additional materials

The electronic version of this doctoral dissertation, as well as additional materials, such as the original forms drawn during the experimentation (in French), the source code and documentation of the RAINBOW Toolkit and its screencast tutorials, can be found on the web site of the Laboratory of Database Applications Engineering (http://info.fundp.ac.be/libd) or on the author's dedicated website (http://www.ramdoyal.be/rainbow).