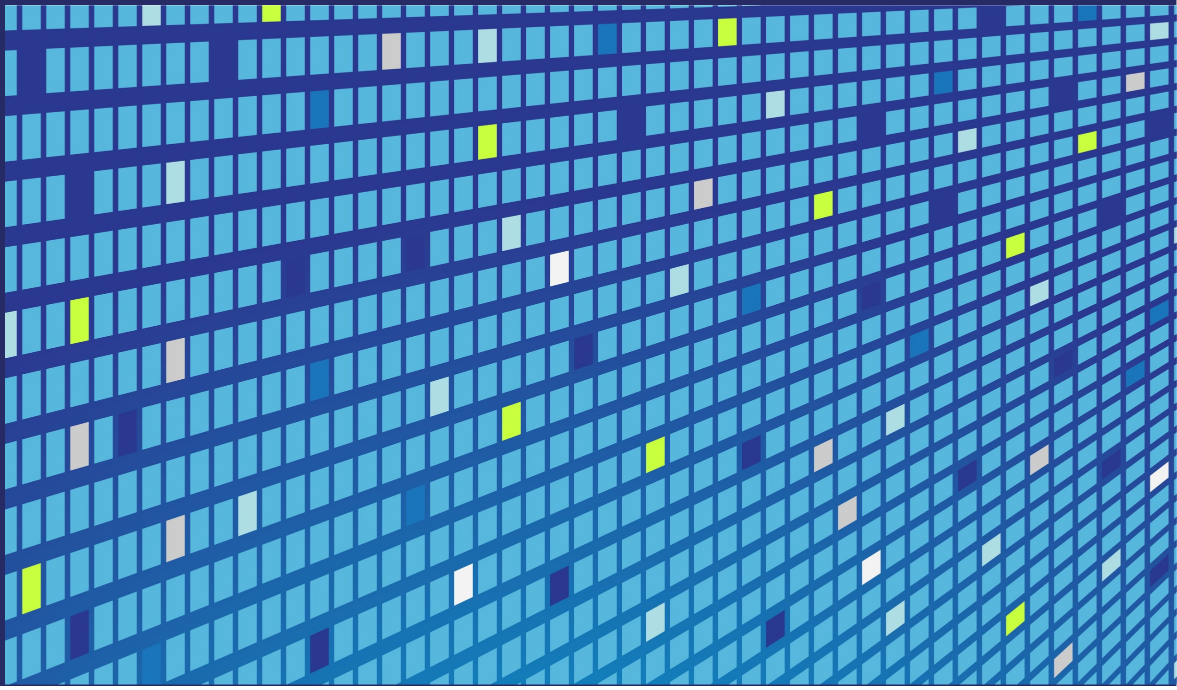
Spreadsheets play an important role in software organizations: they are not only used to define sheets containing data and formulas, but also to collect information from different systems, to perform operations to enrich or simplify data, etc. Unfortunately, spreadsheet systems provide poor support for modularity, abstraction, and transformation, thus making the maintenance of spreadsheets a complex and error-prone task. An emerging solution to handle complex software systems is model-driven engineering. Its basic principle is to consider models as first class entities and to classify any software artifact as either a model or a model instance. In our work, we adapted to spreadsheets several techniques that are inspired by model-driven approaches to generic software systems. In fact, most spreadsheets lack a proper specification or a model. Using reverse engineering techniques we are able to derive various models from legacy spreadsheets: they can be used for several improvements, namely refactoring, safe evolution, migration or even generation of edit assistance. The techniques presented in this book have been integrated in HaExcel, a framework to improve spreadsheet productivity.

Model-based Spreadsheet Engineering



Jácome Cunha

Model-based Spreadsheet Engineering

Using Relational Models to Improve Spreadsheets

Jácome Cunha

Jácome Cunha is a Postdoctoral researcher at the Department of Informatics, University of Minho, Portugal. He obtained his PhD in 2011, following his research on the analysis, transformation, and modeling of spreadsheets. He has co-authored a significant number of scientific publications that have been presented at the best events in his field.

978-3-8484-3036-9

Jácome Cunha

LAP LAMBERT Academic Publishing

Acknowledgments

Four years have passed and I am finally finishing my PhD studies. These have been times of great joy for many reasons: for start, I'm working on something that I like, research. Also during this period I have had the time to be with my friends and family and to enjoy the life with them. The happiness when a paper is accepted in a conference is also quite something. Then, of course, plan the trip and enjoy the conference and the city where it is taking place is another great reason for my happiness.

I think everyone says that to work on a thesis is only possible because we have the support of those we love and that love us back. I will not say the same think just because everyone says it, but because I fill it! Without the support of these people this simply would not be impossible!

First of all I want to thank my supervisor and friend João Saraiva. He took his time to accept me as his PhD student because, at the time, he thought I was a bit insane... Probably he was right. I hope he has changed his mind, though! But then he decided to trust me and that was very important to me. I am not sure if any other person would do such a risky decision, but I am glade he made it. After he accepted to guide me during these times, he was devoted, supportive and a real research and life guide. His research and life experience are quite inspiring. I have no doubts that he is the best supervisor that I could have chosen! Thank you very much for your dedication to me!

Better than having a supervisor is to have two supervisors! The wisdom, incisive comments and brilliant mind of Joost Visser were fundamental for the good results of this work. The fact that he left the department was a great lost to all of us, and specially to me, but that never stopped him to help me during my work. Thank you very much for all what you have done for me!

Martin Erwig came in the end of this work but was quite important. His expertise is inspiring and I know that it will keep motivating me. Thank you for the inspiration!

I want to say thanks to many more people, starting by my office colleagues and

friends (not by any specific order): João Paulo (the one that I keep following), Vilaça (the one always ready for everything), Paulo (the smartest and slowest one), André (the industrial one), Ricardo (the one that do not speak, yells!), Filipe (the father one), Mirandês (the *lesiano* one), João Paulo (the one with a band), Nuno (the geek and paranoiac one), Ana (the geek girl), Xico (the professional driver one), Maia (the one with a TV instead of a monitor), Miguel (the men in the greatcoat), Tiago (the one with almost a Porsche) and Paulo (Jesus himself!). As expected, a very thankful word to *Os Sem Estatuto*. The sharing of ideas, not research related ones, of course, was of great help. Thank you very much!

I would like to thank all my outside work friends. Again in no special oder: Tércio, Marta, Alice, Amadeu, Senhor Engenheiro Malheiro, Raphinha, Xana, Trofa, Zé and Gonçalo. These are the people that are present in the important moments of my life. Somehow, they still keep supporting me! To all of you, thank you very very much for making me happy and enjoying my life!

Now that I'm married, yes I got married during the PhD (nice excuse for some vacations, by the way!), I need to dedicate a paragraph to my wife (otherwise I will have troubles at home!). Actually, I do not need to, but I want to. For many years now, almost a decade, she has been my best friend, my shoulder to cry and basically, my life support machine. Without her encouragement, support and trust it would be impossible to work on something like a PhD. Actually, it would be even difficult to live! Thank you very much for everything! I love you very very much!!!

Finally, I would like to thank my parents: Durante 27 anos tive o incondicional apoio de 2 pessoas, Aristides Cunha e Maria Arminda. Estas são as pessoas que me apoiaram e me ajudaram a seguir em frente em todas as situações da minha vida, não apenas durante o doutoramento, mas também, claro. Pelo vosso amor, que eu sei ser infinito, eu vos agradeço do fundo do meu coração! Amo-vos muito!

UNIÃO EUROPEIA
Fundo Social Europeu

POPH

Some institutions supported the work presented in this thesis. The Portuguese science foundation, Fundação para a Ciência e a Tecnologia, supported this work with a grant (POPH/FSE), `SFRH/BD/30231/2006`, for the whole period of studies. The Department of Informatics and the Computer Science and Technology Center of the University of Minho provided me the conditions for my studies. The ACM SIGPLAN Professional Activities Committee supported some conference participations. Software Improvement Group received me as a guest so I could work closely with Joost.

Model-based Spreadsheet Engineering

Spreadsheets can be viewed as programming languages for non-professional programmers. These so-called “end-user” programmers vastly outnumber professional programmers creating millions of new spreadsheets every year. As a programming language, spreadsheets lack support for abstraction, testing, encapsulation, or structured programming. As a result, and as numerous studies have shown, the high rate of production is accompanied by an alarming high rate of errors. Some studies report that up to 90% of real-world spreadsheets contain errors. After their initial creation, many spreadsheets turn out to be used for storing and processing increasing amounts of data and supporting increasing numbers of users over long periods of time, making them complicated systems.

An emerging solution to handle the complex and evolving software systems is *Model-driven Engineering* (MDE). To consider models as first class entities and any software artifact as a model or a model element is one of the basic principles of MDE.

We adopted some techniques from MDE to solve spreadsheet problems. Most spreadsheets (if not all) lack a proper specification or a model. Using reverse engineering techniques we are able to derive various models from legacy spreadsheets. We use *functional dependencies* (a formalism that allow us to define how some column values depend on other column values) as building blocks for these models. Models can be used for several spreadsheet improvements, namely refactoring, safe evolution, migration or even generation of edit assistance. The techniques presented in this work are available under the framework HAEXCEL that we developed. It is composed of online and batch tools, reusable HASKELL libraries and *OpenOffice.org* extensions.

A study with several end-users was organized to survey the impact of the techniques we designed. The results of this study indicate that the models can bring great benefits to spreadsheet engineering helping users to commit fewer errors and to work faster.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Summary

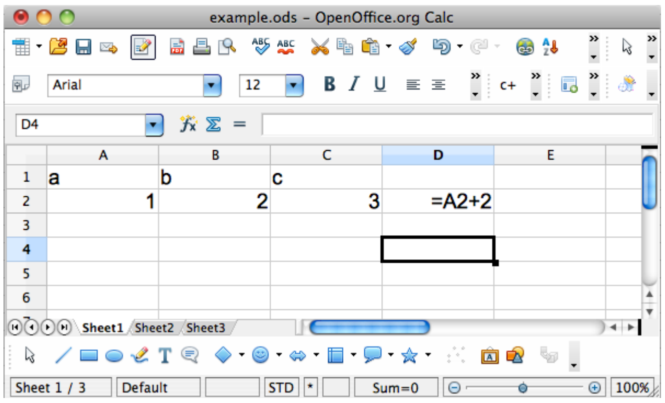
In this chapter we present a small introduction to the concepts that will be studied throughout this thesis. In particular, we introduce spreadsheets, using a simple example. The problems inherent to spreadsheets are described, as well as the solutions we propose to help solving them.

1.1 Spreadsheets

For many people, the programming language of choice is a spreadsheet. This is particularly true for non-professional programmers, often defined as “end-user” programmers (Nardi 1993). An end-user programmer is typically a teacher, an engineer, a physicist, a secretary, an accountant or a manager. In fact, almost every person but a trained programmer is considered an end user. End users’ interest in computer programming is usually limited to get a concrete task done; often they are not interested in programming *per se*.

End-user programmers outnumber professional programmers, and this difference is projected to increase more rapidly. In fact, as a study performed in 2005 shows, in the U.S. alone, the number of end-user programmers is conservatively estimated at 11 million, compared to only 2.75 million other, professional programmers (Scaffidi *et al.* 2005). These facts suggest that the spreadsheet, which is a widely used and commercially successful end-user programming language, is also a particularly significant target for the broader application of programming-language design principles.

A spreadsheet is a digital document created with a specific software application that simulates a paper displaying multiple cells that together make up a grid consisting of rows and columns, each cell containing alphanumeric text, numeric values or formulas. A formula defines how the content of that cell is calculated from the contents of other cells and it is updated each time those cells are changed. Constant values are also accepted as formula parameters. For example, a formula can be used to sum all the cells of a particular column. Moreover, a cell can be defined by a reference to another cell, thus displaying the referenced cell content. One of the nicest features of spreadsheets is their ability to incrementally recalculate the entire sheet automatically after a change to a single cell is made. Figure 1.1 illustrates a simple spreadsheet.


example.ods – OpenOffice.org Calc

D4

	A	B	C	D	E
1	a	b	c		
2	1	2	3	=A2+2	
3					
4					
5					
6					

Figure 1.1: Example of a spreadsheet.

In a spreadsheet, cells are typically identified by the pair column (usually a capital letter) and row (usually a number). In the spreadsheet shown in Figure 1.1 the value in cell `A1` is the constant `a` (and alphanumeric text) while cell `C3` contains the numeric value `3`. Cell `D4` is a formula since it starts with the $=$ sign. The formula adds `2` to the value in cell `A2`, which is the formula is represented by the *reference* to such cell trough the text `A2`.

Historically, *Visicalc* (O'Donovan 1984) is usually considered the first electronic spreadsheet system. Later, *Lotus 1-2-3* (Bookbinder 1989) led the spreadsheet market when *MS-DOS*[1] was the dominant operating system. Nowadays, it is *Excel* (Campbell 1985; O'Leary 2008) that has the largest market share both on *Windows* and *Macintosh* platforms. The component *Calc* from the *OpenOffice.org* suite (Riley 2009) can be seen as an open source alternative to *Excel*.

The advent of the internet is changing the way people interact with computers influencing our lifestyle. Internet is also playing an important role in the context of

[1]The initial version of Microsoft operating system for IBM personal computers.

spreadsheets. Indeed, spreadsheet systems usually seen as desktop applications are becoming web-based applications. *Google Docs* (Google 2011) is one of the first examples of a web-based spreadsheet system that is also influencing other systems like the widely used *Excel* from Microsoft. As a consequence, spreadsheet systems will soon be widely available in our mobile devices, like mobile phones and tablet computers.

In spite of their huge popularity, spreadsheets still have some problems, specially when considered from the point of view of a programming language. The purpose of the next section is to describe some of these problems.

1.2 Problem Statement

Spreadsheet systems offer a high level of flexibility, making it easy for people to start working with them. The downside of this freedom is that it also offers ample opportunity to create erroneous spreadsheets. In fact, numerous studies report that up to 90% of real-world spreadsheets contain errors (Panko 2000; Rajalingham *et al.* 2001; Powell and Baker 2003; EuSpRIG 2011). This happens because, as programming systems, spreadsheets lack the support provided by modern programming languages/environments, namely:

Abstraction The following definition of abstraction, extracted from (Oxford 2011), should help the reader to understand how this term applies to computer science:

Abstract: existing in thought or as an idea but not having a physical or concrete existence.

This definition can easily be adapted to computer science: an abstraction is a general, non-concrete concept or idea; it is a classification of instances or objects.

A good example of an abstraction is the one of *classes* in the object-oriented (OO) programming paradigm (Meyer 1997). There, it is possible to define a class representing, for example, *animals*. In such class, only the characteristics of all animals are represented; one does not specify which animal or even the kind of animal it represents. Then, concrete instance of this class can be defined for concrete animals. Unfortunately, this is not possible to accomplish in a spreadsheet environment. It is only possible to define concrete values, not abstract concepts.

Encapsulation Booch *et al.* (2007) defined encapsulation as "*the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation*".

The motivation for encapsulation is to achieve higher potential for change accommodation. It should be possible to improve the internal mechanisms of a component without this having impact on others. Encapsulation also protects the integrity of a component by preventing users from directly and without control changing its internal state. Encapsulation also reduces a system complexity and thus increases its robustness since it limits the interdependencies between software components (Booch *et al.* 2007).

Encapsulation is usually achieved by creating some kind of capsule, that is, some kind of component, possibly with an internal state, which can only be accessed by controlled methods. This is, in general, very difficult, if not impossible, to achieve in a spreadsheet environment.

Type system A type system associates types with each computed value. It may be defined as "*a tractable syntactic framework for classifying phrases according to the kinds of values they compute*" (Pierce 2002). By examining the flow of these values, a type system attempts to prove that no type errors can occur. The type system in question determines what constitutes a type error, but a type system generally seeks to guarantee that operations expecting a certain kind of value are not used with values for which that operation does not make sense.

Although spreadsheets have a very basic type system (they can distinguish, for example, numbers and strings), they cannot detect when, for example, the user is trying to sum shoes and ties (in a cloth stock application, for instance). The exception to this is the work made by Abraham *et al.* on a system to infer unit and header information for spreadsheets (Abraham and Erwig 2004).

Testing Software testing is a technique used to detect software failures so that defects may be discovered and corrected. Unfortunately, this is a non-trivial task for software in general. In fact, "*program testing can be used to show the presence of bugs, but never to show their absence*" (Dijkstra 1970).

Software testing often includes examination of code as well as its execution in various environments and conditions. Moreover, information derived from software testing may be used to correct the process by which the software is developed (Huizinga and Kolawa 2007).

The fact that data and computations are all on the same level on a spreadsheet makes it difficult to effectively test them. As a consequence, it is difficult to distinguish input data and computations since there is not a way to clearly define them. An exception to the lack of testing techniques for spreadsheets can be found in (Abraham and Erwig 2006c).

The lack of these important features results in error-prone spreadsheets. Errors are made during the creation of a spreadsheet as well as when it is modified. The problem gets exacerbated when the user modifying the spreadsheet does not fully understand its functionality. These factors make widespread reuse of spreadsheets difficult and prone to errors.

Given this scenario, and to help researchers improving it, Panko and Aurigemma (2010) proposed a spreadsheet error taxonomy as illustrated in Figure 1.2.

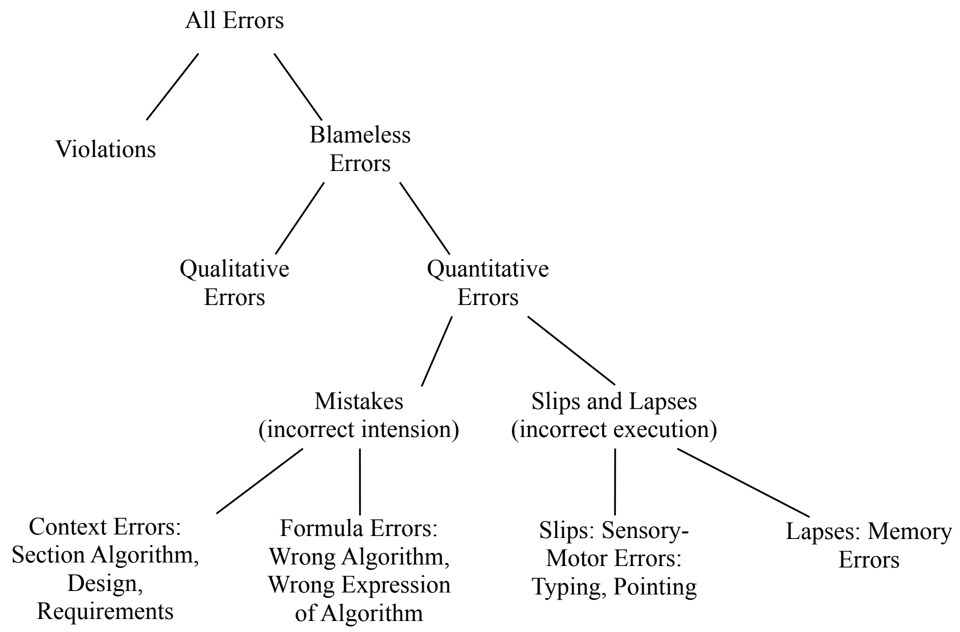

All Errors
Violations
Blameless Errors
Qualitative Errors
Quantitative Errors
Mistakes (incorrect intension)
Slips and Lapses (incorrect execution)
Context Errors: Section Algorithm, Design, Requirements
Formula Errors: Wrong Algorithm, Wrong Expression of Algorithm
Slips: Sensory-Motor Errors: Typing, Pointing
Lapses: Memory Errors

Figure 1.2: Spreadsheet error taxonomy.

Some of these errors have high impact in company productivity (Croll 2007, 2009) leading to companies and institutions losing millions of dollars (EuSpRIG 2011). In fact, the Jamaican Banking System collapsed in its entirety in the late 1990's partly due to the use of spreadsheets and a consequent failure to manage and control them (Lemieux 2002, 2008).

In the following section we will briefly present existing approaches to solve some of the problems related to spreadsheets.

1.3 Some Possible Solutions

As explained in the previous section, there are several problems with spreadsheets. Several researchers have realized the importance of spreadsheets and have presented several approaches to reduce the incidence of errors in them. These approaches can be divided in two main categories: (1) solutions not directly related to end users, but more indicated for professionals and (2) solutions directly shaped for spreadsheet end users, helping them closely. Category (1) can be subdivided as follows:

1. Recommendations for better spreadsheet design (Ronen *et al.* 1989; Yoder and Cohn 1994; Isakowitz *et al.* 1995; Rajalingham *et al.* 2000; Powell and Baker 2003).

2. Auditing spreadsheets to detect and remove errors (Panko 1999; Sajaniemi 2000; Mittermeir and Clermont 2002).

Although the techniques from these two subcategories can be used by end users, they are more indicated to professionals. Recommendations can be taken into account by a professional programmer developing a spreadsheet (that will be then used by an end user). Auditing techniques are in principle used by professional to assess spreadsheets.

On the other hand, category (2) intends to produce techniques to help end users in a daily base, that is, when actually working with spreadsheets:

1. Testing (Rothermel *et al.* 2001; Fisher *et al.* 2002; Burnett *et al.* 2002).

2. Automatic consistency checking (Erwig and Burnett 2002; Ahmad *et al.* 2003; Burnett *et al.* 2003; Antoniu *et al.* 2004; Abraham and Erwig 2004; Coblenz *et al.* 2005).

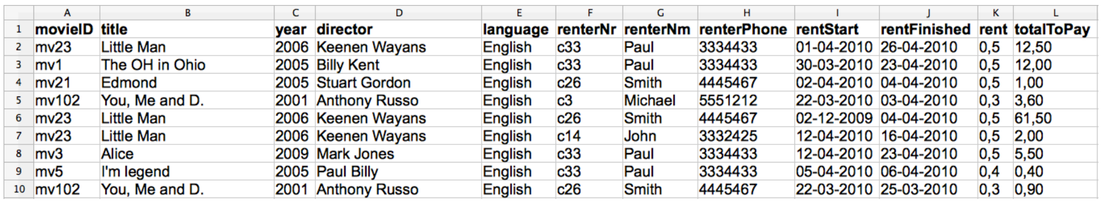
3. Error prevention techniques (Erwig *et al.* 2005; Engels and Erwig 2005; Cunha *et al.* 2009a,b; Hermans *et al.* 2010; Cunha *et al.* 2010; Beckwith *et al.* 2011a,b; Cunha *et al.* 2011).

The first two subcategories, testing and consistency checking, should be used by users to detect errors in their spreadsheets, that is, they give the users some feedback on their errors. Error preventing is intended to help users not creating errors. This is a technique that can be used before testing and checking. In fact, this should reduce the need for testing.

Our approach is intended to work when users are working with a spreadsheet, but in a preventive way, that is, we fit in the error prevention techniques. In the following section we will present our solution.

1.4 Our Solution - An Example

In order to better present our work we shall consider the following example adapted from (Berdaguer *et al.* 2007) and modeled in a spreadsheet as shown in Figure 1.3.



	A	B	C	D	E	F	G	H	I	J	K	L
1	**movieID**	**title**	**year**	**director**	**language**	**renterNr**	**renterNm**	**renterPhone**	**rentStart**	**rentFinished**	**rent**	**totalToPay**
2	mv23	Little Man	2006	Keenen Wayans	English	c33	Paul	3334433	01-04-2010	26-04-2010	0,5	12,50
3	mv1	The OH in Ohio	2005	Billy Kent	English	c33	Paul	3334433	30-03-2010	23-04-2010	0,5	12,00
4	mv21	Edmond	2005	Stuart Gordon	English	c26	Smith	4445467	02-04-2010	04-04-2010	0,5	1,00
5	mv102	You, Me and D.	2001	Anthony Russo	English	c3	Michael	5551212	22-03-2010	03-04-2010	0,3	3,60
6	mv23	Little Man	2006	Keenen Wayans	English	c26	Smith	4445467	02-12-2009	04-04-2010	0,5	61,50
7	mv23	Little Man	2006	Keenen Wayans	English	c14	John	3332425	12-04-2010	16-04-2010	0,5	2,00
8	mv3	Alice	2009	Mark Jones	English	c33	Paul	3334433	12-04-2010	23-04-2010	0,5	5,50
9	mv5	I'm legend	2005	Paul Billy	English	c33	Paul	3334433	05-04-2010	06-04-2010	0,4	0,40
10	mv102	You, Me and D.	2001	Anthony Russo	English	c26	Smith	4445467	22-03-2010	25-03-2010	0,3	0,90

Figure 1.3: A spreadsheet representing a movie renting system.

This spreadsheet represents a movie renting system containing information about movies, renters and the leases themselves. Columns `A` to `E` contain information about the movies, columns `F` to `H` contain information about the renters and the remaining columns contain information about the leases (the labels of the columns should be self explanatory).

Having these spreadsheet data, we wish to understand its *business model*, that is, the semantic of the spreadsheet, its logic. For this simple and small example it is possible for an experienced programmer to define the business model underlying the spreadsheet data. However, this is difficult for an end user, as shown by Abraham and Erwig (2006a). As a consequence, we wish to define techniques to infer the business logic model automatically with no end-user interaction.

Functional Dependencies

The interdependencies between spreadsheet data can be captured by a powerful mechanism called *functional dependencies*, a concept from relational databases theory describing the relationship between attributes of a table/relation (Codd 1970).

Let us take as an example the following subset of attributes of the movies example: $\{renterNr, movieID, rentStart, rentFinish\}$. Even non-experts in movie renting systems will accept the following "business" rule: *the start and end date of renting and the rent of a movie determine the total amount to pay*. This restriction is an example of a so-called *functional dependency* among attributes, which can be stated more formally as follows: *attribute totalToPay is functionally dependent on rentStart, rentFinish and rent*. In the standard practice, this will be abbreviated by writing

$$rentStart, rentFinish, rent \rightharpoonup totalToPay$$

which has the following, alternative reading: *rentStart, rentFinish and rent functionally determine totalToPay*. The left-hand side of the harpoon is termed *antecedent* of the functional dependency and the right-hand side *consequent*. Note that the consequent can be an empty set, meaning that no repetition of antecedent values is allowed.

A second example of a functional dependency is as follows:

$$renterNr \rightharpoonup renterNm$$

This dependency encodes the fact that the code of a renter, that is, its *renterNr* determines its name. In other words this means that cannot exist two different codes for two different renters. Note that there can exist two clients with the same name, that is, the dependency $renterNm \rightharpoonup renterNr$ does not hold.

By reasoning on the spreadsheet data we can discover the functional dependencies induced by it. These techniques are usually applied in the database realm which make them not very suitable for spreadsheets.

In our work we refine these techniques to better work with data from spreadsheets using several heuristics to filter out the "accidental" dependencies, that is, the dependencies that are in the data but do not reflect any real relationship. This may happen if the data does not represent well enough the situation. For example, if our data would have only one renter it would be impossible to see the relationship between its number and its name. For the movies example, the functional dependency $renterNm \rightharpoonup language$ exists, meaning that we can determine the language of a movie based on a renter's name. This happens because the language of all movies included in the spreadsheet is `English`. As a consequence, not only does *renterNm* determine the language, but also most of the other columns determine it. These dependencies are unusable and should be discarded from further consideration. For our running example, we would like to infer the following dependencies:

$$\begin{array}{l}
\mathit{language} \rightharpoonup \{\} \\
\mathit{rentStart}, \mathit{rentFinish}, \mathit{rent} \rightharpoonup \mathit{totalToPay} \\
\mathit{renterNr} \rightharpoonup \mathit{renterNm}, \mathit{renterPhone} \\
\mathit{movieID} \rightharpoonup \mathit{title}, \mathit{year}, \mathit{director}, \mathit{rent} \\
\mathit{language}, \mathit{rentStart}, \mathit{rentFinish}, \mathit{renterNr}, \mathit{movieID} \rightharpoonup \{\}
\end{array}$$

These dependencies reflect the entities present in this system: *languages of movies*, *payments*, *renters*, *movies* and *leases themselves*.

Inferring Models for Spreadsheets

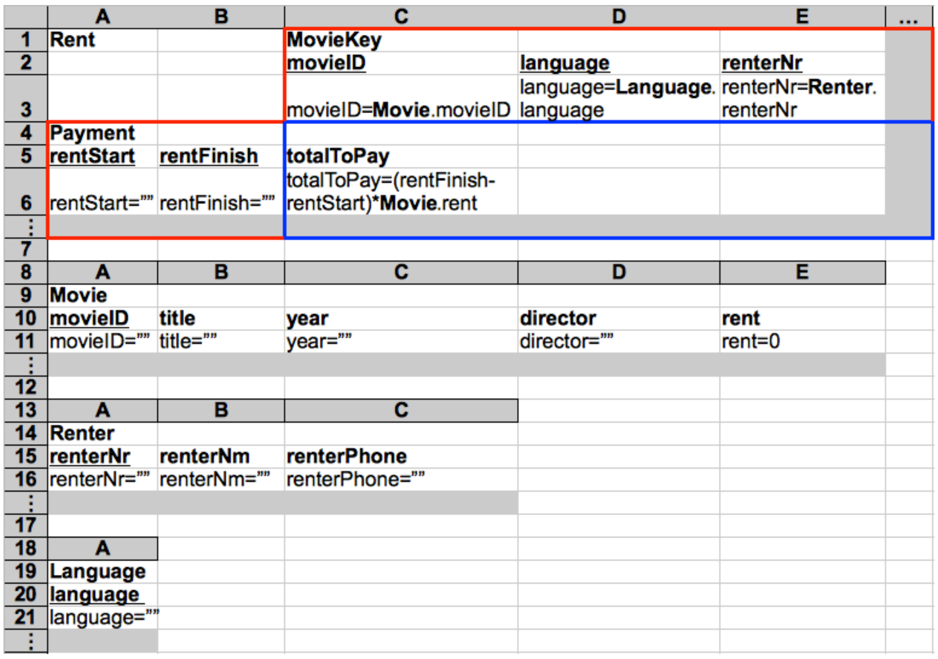
From the functional dependencies previously defined we would like to compute a normalized relational database schema (Codd 1970) reflecting the restrictions imposed by the dependencies. The following schema is a possible solution:

$$\begin{array}{ll}
\mathit{Language} & (\underline{\mathit{language}}) \\
\mathit{Payment} & (\underline{\mathit{rentStart}, \mathit{rentFinish}, \mathit{rent}}, \mathit{totalToPay}) \\
\mathit{Renter} & (\underline{\mathit{renterNr}}, \mathit{renterNm}, \mathit{renterPhone}) \\
\mathit{Movie} & (\underline{\mathit{movieID}}, \mathit{title}, \mathit{year}, \mathit{director}, \mathit{rent}) \\
<\mathit{Rent}> & (\underline{\#\mathit{language}, \#\mathit{rentStart}, \#\mathit{rentFinish}, \#\mathit{renterNr}, \#\mathit{movieID}})
\end{array}$$

This schema has five tables implementing the five entities we have identified above. Notice that the underlined attributes are known as the *primary keys* of the tables, that is, the columns that uniquely identify each row of a table. The symbol # marks the *foreign key* columns, that is, the columns that are references to other columns.

From the inferred relational model, we would also like to devise a *ClassSheet* model (Engels and Erwig 2005). *ClassSheet* models are a formalism to define the model of a spreadsheet, that is, its business rules. A spreadsheet application consistent with the model could be automatically generated, and thus, a large variety of errors could be prevented. Figure 1.4 illustrates the desired model.

Given the similarity between *ClassSheet* models and *Unified Modeling Language* (UML) class diagrams (Rumbaugh *et al.* 2004), we would like to generate class dia-



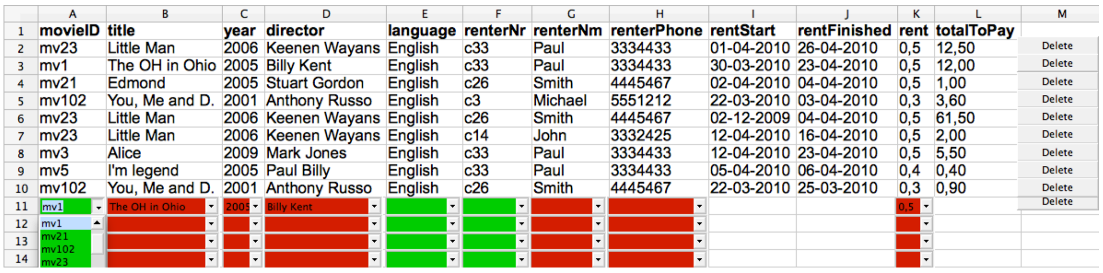
	A	B	C	D	E	...
1	**Rent**		**MovieKey**			
2			**movieID**	**language**	**renterNr**	
3			movieID=**Movie**.movieID	language=**Language**.language	renterNr=**Renter**.renterNr	
4	**Payment**					
5	**rentStart**	**rentFinish**	**totalToPay**			
6	rentStart=""	rentFinish=""	totalToPay=(rentFinish-rentStart)***Movie**.rent			
⋮						
7						
8	A	B	C	D	E	
9	**Movie**					
10	**movieID**	**title**	**year**	**director**	**rent**	
11	movieID=""	title=""	year=""	director=""	rent=0	
⋮						
12						
13	A	B	C			
14	**Renter**					
15	**renterNr**	**renterNm**	**renterPhone**			
16	renterNr=""	renterNm=""	renterPhone=""			
⋮						
17						
18	A					
19	**Language**					
20	**language**					
21	language=""					
⋮						

Figure 1.4: *ClassSheet* model representing a movie renting system.

grams from *ClassSheets*. These diagrams could then be used to support other migrations, for example, to the object-oriented paradigm.

Edit Assistance for Spreadsheets

Using the functional dependencies inferred before we would like to generate a new spreadsheet similar to the original one, but with edit assistance, that is, with some kind of help to the user when editing the spreadsheet. Figure 1.5 illustrates such an environment.



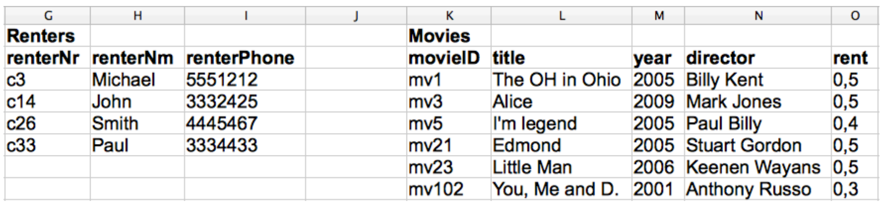
	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**movieID**	**title**	**year**	**director**	**language**	**renterNr**	**renterNm**	**renterPhone**	**rentStart**	**rentFinished**	**rent**	**totalToPay**	
2	mv23	Little Man	2006	Keenen Wayans	English	c33	Paul	3334433	01-04-2010	26-04-2010	0,5	12,50	Delete
3	mv1	The OH in Ohio	2005	Billy Kent	English	c33	Paul	3334433	30-03-2010	23-04-2010	0,5	12,00	Delete
4	mv21	Edmond	2005	Stuart Gordon	English	c26	Smith	4445467	02-04-2010	04-04-2010	0,5	1,00	Delete
5	mv102	You, Me and D.	2001	Anthony Russo	English	c3	Michael	5551212	22-03-2010	03-04-2010	0,3	3,60	Delete
6	mv23	Little Man	2006	Keenen Wayans	English	c26	Smith	4445467	02-12-2009	04-04-2010	0,5	61,50	Delete
7	mv23	Little Man	2006	Keenen Wayans	English	c14	John	3332425	12-04-2010	16-04-2010	0,5	2,00	Delete
8	mv3	Alice	2009	Mark Jones	English	c33	Paul	3334433	12-04-2010	23-04-2010	0,5	5,50	Delete
9	mv5	I'm legend	2005	Paul Billy	English	c33	Paul	3334433	05-04-2010	06-04-2010	0,4	0,40	Delete
10	mv102	You, Me and D.	2001	Anthony Russo	English	c26	Smith	4445467	22-03-2010	25-03-2010	0,3	0,90	Delete
11	🟩 mv1	🟥 The OH in Ohio	🟥 2005	🟥 Billy Kent	🟩	🟩	🟥	🟥			🟥 0,5		Delete
12	mv1	🟥	🟥	🟥	🟩	🟩	🟥	🟥			🟥		
13	mv21 mv102	🟥	🟥	🟥	🟩	🟩	🟥	🟥			🟥		
14	mv23	🟥	🟥	🟥	🟩	🟩	🟥	🟥			🟥		

Figure 1.5: A spreadsheet representing a movie renting system with edit assistance.

Since we know the relationship between columns, given by the functional dependencies, we would like to use them to create a mechanism that automatically fills in some columns. The columns with green[2] *combo boxes* represent antecedents in functional dependencies and the red ones, represent consequents. Since we know that antecedents uniquely determine consequents, when the user selects a value in a green cell, the corresponding red cells are automatically filled in. Such a mechanism is present in the spreadsheet illustrated in Figure 1.5: when the user selected a possible value for the *movieID* column (namely `mv1`), columns *title*, *year*, *director* and *rent* were automatically filled in.

Spreadsheet Refactoring

From the relational schema previously inferred, we would like to create a new spreadsheet without data redundancy, that is, without having data repeated several times. Moreover, this spreadsheet should respect the relational schema. Figure 1.6 represents part of a possible refactored movie renting system spreadsheet.



G	H	I	J	K	L	M	N	O
Renters				**Movies**				
renterNr	**renterNm**	**renterPhone**		**movieID**	**title**	**year**	**director**	**rent**
c3	Michael	5551212		mv1	The OH in Ohio	2005	Billy Kent	0,5
c14	John	3332425		mv3	Alice	2009	Mark Jones	0,5
c26	Smith	4445467		mv5	I'm legend	2005	Paul Billy	0,4
c33	Paul	3334433		mv21	Edmond	2005	Stuart Gordon	0,5
				mv23	Little Man	2006	Keenen Wayans	0,5
				mv102	You, Me and D.	2001	Anthony Russo	0,3

Figure 1.6: Refactored spreadsheet representing a movie renting system.

This spreadsheet does not contain data redundancy as opposed to the one presented in Figure 1.3, where, for example, the phone number of `Smith` appears three times.

Spreadsheet Migration

Spreadsheets are applications usually created by one end user, without planning ahead of time for maintainability or scalability. Still, after their initial creation, many spreadsheets turn out to be used for storing and processing increasing amounts of data and supporting increasing numbers of users over long periods of time. To turn such spreadsheets into database-backed multi-user applications with high maintainability is not a smooth transition, but requires substantial time and effort.

Having inferred the relational database schema and the data in the same format, a smooth transition from spreadsheets to relational databases is possible. In fact, we would like to be able to generate *Structured Query Language* (Date 1986) scripts to

[2]Colors are visible through the digital version of this document.

create and populate a real relational database. Using *data refinement* theory, the transformation between spreadsheets and databases can be achieved. *Data refinement* is the systematic substitution of one data type by another in a program. Usually, the new data type is more efficient than the old, but possibly more complex (Morgan and Gardiner 1990; Oliveira 1990, 2008). Rules between the tabular spreadsheet model and the relational schema are capable of the transformation between the two models. Moreover, we would like to define the functions that migrate the data between these two data representations. Again, we use data refinement theory so that we can get for free the functions to migrate the data. From this result it is easy to generate the desired scripts.

Evolution of Spreadsheets

The transformation of a spreadsheet into another one with the same data but another layout is a particular case of an evolution step (Lämmel and Lohmann 2001). Using the *ClassSheet* model presented in Figure 1.4 we would like to define a set of evolution steps that represent usual updates that users do, for example, inserting a new column or moving part of the spreadsheet to another sheet. When changing a spreadsheet affects the underlying model the probability of making an error is quite high. We would like to create several rules that allow this evolution in a safe way.

The steps described in this section can be generalized and automatized for general spreadsheets. During this thesis we will describe in detail how this can be done. In fact, the process described in this section for the movies example was produced by our techniques.

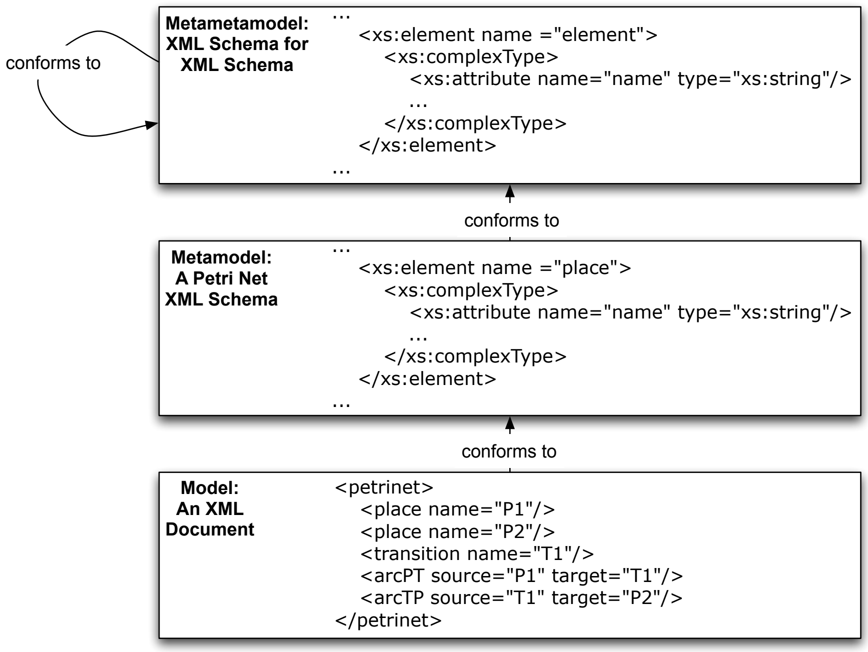
In the following section we will review our solution from a *Model-Driven Engineering* (MDE) point of view. We will show that, in fact, we have followed the principles from this field, which are well established and an active area of research.

1.5 Reviewing Our Solution

In this section we briefly introduce *Model-driven Engineering* (Stahl *et al.* 2006) and explain how our approach follows this successful approach.

In the last years MDE has emerged as a solution to handle complex and evolving software systems. To consider models as first class entities and any software artifact as a model or as a model element is one of the basic principles of MDE.

Consider as an example XML (XML 2008). An XML document can be seen as a model while its schema (or DTD) as its metamodel. Figure 1.7 illustrates a three level structure in an XML environment (Bézivin 2006).


Metametamodel: XML Schema for XML Schema
conforms to
conforms to
Metamodel: A Petri Net XML Schema
conforms to
Model: An XML Document

```
...
  <xs:element name ="element">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string"/>
      ...
    </xs:complexType>
  </xs:element>
...
```

```
...
  <xs:element name ="place">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string"/>
      ...
    </xs:complexType>
  </xs:element>
...
```

```
<petrinet>
  <place name="P1"/>
  <place name="P2"/>
  <transition name="T1"/>
  <arcPT source="P1" target="T1"/>
  <arcTP source="T1" target="P2"/>
</petrinet>
```

Figure 1.7: A three level structure in an XML environment (Bézivin 2006).

The approach we follow in this work is closely related to the MDE principles. In fact, we consider spreadsheets as models and work with three kinds of metamodels to specify spreadsheets: relational schemas, *ClassSheet* models and UML class diagrams. Consider the illustration of our approach in Figure 1.8.

If we look at it from left to right, we can see that from the original spreadsheet we can automatically infer functional dependencies representing the relationship between the spreadsheet data. By reasoning with these functional dependencies we can infer a relational database schema. These schemas are used in the database realm to model the logic of databases. We can automatically infer this kind of models from existing spreadsheets so they can have some specification of their business logic. From these models we can generate refactored spreadsheets that are normalized (from a data point of view), *Refactored SS* in the right-hand side of Figure 1.8. Moreover, from these models we can generate scripts to automatically create and populate relational databases (below the refactored spreadsheet). This kind of behavior is common in the MDE approach because it produces good results. In fact, it is quite common to

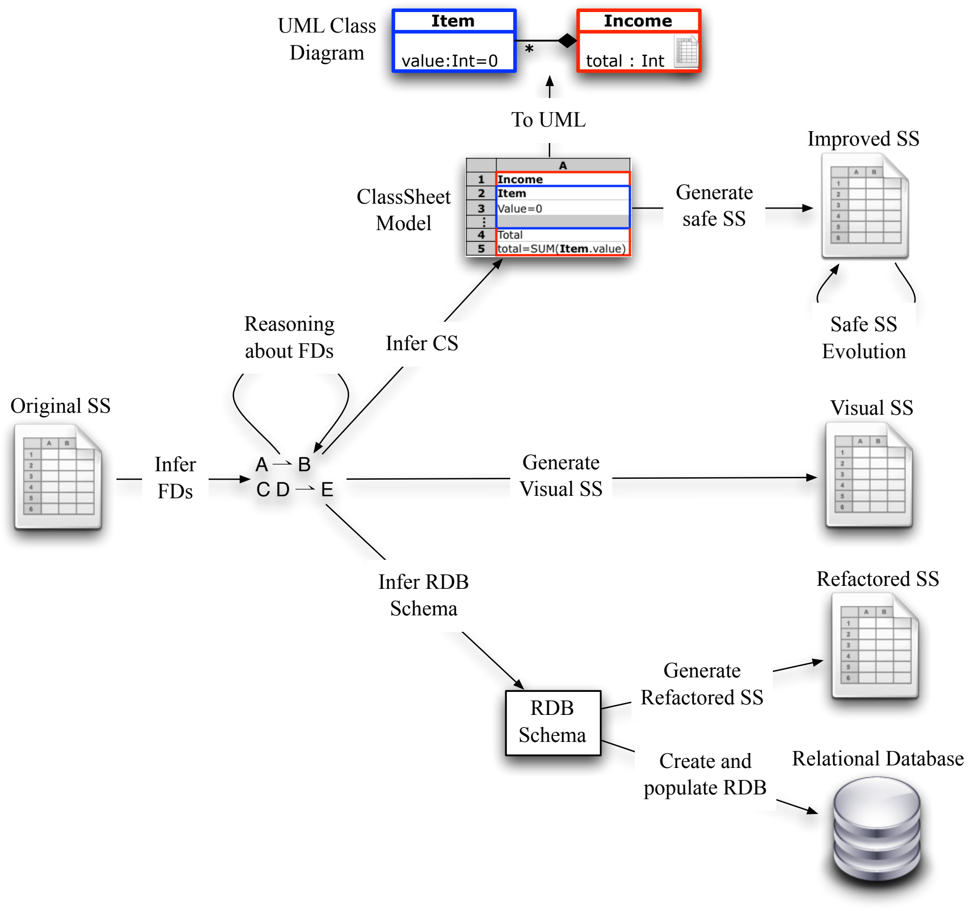


Figure 1.8: Overview of the work presented in this thesis.

generate code from models, for example, to generate object-oriented code from UML specifications.

Relational schemas are quite appropriate to model spreadsheets in a data organization and manipulation way, but they miss some important characteristics from spreadsheets like, for example, spatial constraints. On the other hand, *ClassSheets* allow us to express business object structures within a spreadsheet using concepts from UML. Moreover, *ClassSheet* models allow us to completely specify a spreadsheet. The reader can see in the overview figure that we can automatically generate *ClassSheets* from spreadsheets. In fact, a spreadsheet application consistent with the model can be automatically generated, and thus, a large variety of errors can be prevented. This generated version allows only a finite and controlled set of actions, each of them respecting the model. This is represented in Figure 1.8 as *Improved SS* on the right-hand side. Once more, this is the kind of behavior one would expect when working in any MDE setting. The *ClassSheet* model can also be used to generate a more generic model: a UML class diagram (above the *ClassSheet* model in the figure). This allows further transformations to others paradigms using already existing techniques from the UML realm.

A final possible transformation in our work is the generation of a spreadsheet with edit assistance for the user (*Visual SS* in Figure 1.8). This helps the user introducing data that conforms the previously inferred set of functional dependencies. Once more, this kind of controlled editing approach is based on MDE principles.

We have presented an overview of the solution we propose to improve the productivity of spreadsheets end users. In the next section we will state the research questions we intend to answer with our work.

1.6 Research Questions

In previous sections we have presented some of the problems related to spreadsheets. We have also shown the solution we engineered to solve some of these issues. Next, we list the research questions we will try to answer with this thesis.

RQ1 Can we automatically infer the implicit logic of a spreadsheet and produce a specification or model describing it?

RQ2 Can we use these specifications/models to improve spreadsheet environments in such a way that end users commit fewer errors, that is, can these models prevent users to commit errors?

RQ3 To which extent can we create specifications for spreadsheets and improve them in a non-invasive way?

In the end of this thesis we will give clear answers to these questions.

1.7 Contributions

With the work presented in this thesis we have made several contributions in different areas such as programming languages or software engineering. A summary of these contributions is presented next:

- We study and present techniques to infer and reason about functional dependencies in the context of spreadsheets;
- Using the idiosyncrasies of spreadsheets, we present techniques to automatic inference of relational schemas for spreadsheets, *ClassSheet* models and UML class diagrams;
- Using functional dependencies we can infer edit assistance for spreadsheets including, for example, the auto-completion of some columns;
- We calculate the formal relationship between spreadsheet models and relational schemas. Rules for the migration between these two fields are presented;
- Based on a relational schema we are able to produce a new spreadsheet that is more organized than the original one and thus better for handling data;
- We improve the 2LT framework (Cunha *et al.* 2006; Alves *et al.* 2008; Visser 2008) to support spreadsheet models/specifications (based on *ClassSheet* models). We also develop a series of common evolution steps for spreadsheets including, for example, insertion of a column in each instance of a model;
- A study with end users validating the results of our work is presented;
- All the techniques here presented are available under an open source framework, HAEXCEL, that can be reused in other projects.

For a more extensive description of our contributions the reader is referred to Chapter 9.

1.8 Structure of the thesis

This thesis is organized as follows:

Chapter 2 presents techniques to derive functional dependencies for spreadsheets. The filtering and normalization of those dependencies is also presented.
This chapter is based on the following paper:
Jácome Cunha, Martin Erwig, João Saraiva. Automatically inferring ClassSheet models from spreadsheets. VL/HCC '2010. 233–241.

Chapter 3 presents techniques to derive models from functional dependencies for spreadsheets. Relational database schemas, *ClassSheets* and UML diagrams can be automatically generated.
This chapter is based on the following papers:
Jácome Cunha, João Saraiva, Joost Visser. From spreadsheets to relational databases and back. PEPM '2009. 179–188.
Jácome Cunha, Martin Erwig, João Saraiva. Automatically inferring ClassSheet models from spreadsheets. VL/HCC '2010. 233–241.

Chapter 4 explains how to use the functional dependencies induced in Chapter 2 to generate edit assistance for spreadsheets.
This chapter is based on the following paper:
Jácome Cunha, João Saraiva, Joost Visser. Discovery-based edit assistance for spreadsheets. VL/HCC '2009. 233–237.

Chapter 5 presents methods to migrate spreadsheets to relational databases. The inverse process is also explained.
This chapter is based on the following paper:
Jácome Cunha, João Saraiva, Joost Visser. From spreadsheets to relational databases and back. PEPM '2009. 179–188.

Chapter 6 shows how to specify spreadsheet models in the 2LT platform and how to use this framework to do evolution of spreadsheet models and instances.
This chapter is based on the following paper:
Jácome Cunha, Joost Visser, Tiago Alves, João Saraiva. Type-safe evolution of spreadsheets. FASE '2011. 186–201.

Chapter 7 presents an evaluation of the work here exposed. In this evaluation we present an empirical study with human subjects using our techniques.
This chapter is based on the following papers:
Laura Beckwith, Jácome Cunha, João Paulo Fernandes, João Saraiva. End-users productivity in model-based spreadsheets: an empirical study. IS-EUD '2011. 282–288.

Laura Beckwith, Jácome Cunha, João Paulo Fernandes, João Saraiva. An empirical study on end-users productivity using model-based spreadsheets. EuSpRiG '2011. 87–100.

Chapter 8 presents the framework developed to support all the techniques introduced in this work, HaExcel.
This chapter is based on the papers referenced on the other chapters.

Chapter 9 exposes our conclusions and future research directions.

Chapter 2

Functional Dependencies for Spreadsheets

Summary

The inference of functional dependencies is a data mining process used in databases, and thus, not very suitable for other paradigms.
In this chapter we present an algorithm to extract functional dependencies from spreadsheet data. We investigate how the idiosyncrasies of spreadsheets can be exploited to infer functional dependencies that well characterize the business model of the underlying spreadsheet. For example, the order of columns and the semantic of their labels are considered and used to produce more realistic functional dependencies. Moreover, formulas also induce functional dependencies.
These dependencies can then be normalized so they can help preventing data redundancy, for example.

2.1 Introduction

Spreadsheets are applications often created by one end user, without planning ahead of time for maintainability or scalability. Still, after their initial creation, many spreadsheets turn out to be used to store and process increasing amounts of data and support increasing numbers of users over long periods of time. To turn such spreadsheets into database-backed multi-user applications with high maintainability is not a smooth transition. In fact, it requires substantial time and effort.

In this chapter, we develop techniques to discover functional dependencies in existing spreadsheets. These dependencies will be the building blocks for further transformations, for example, to migrate spreadsheet to database. Figure 2.1 illustrates the steps necessary to infer functional dependencies from spreadsheets. Note that, octagonal figures represent computations, and squares, results from those computations.

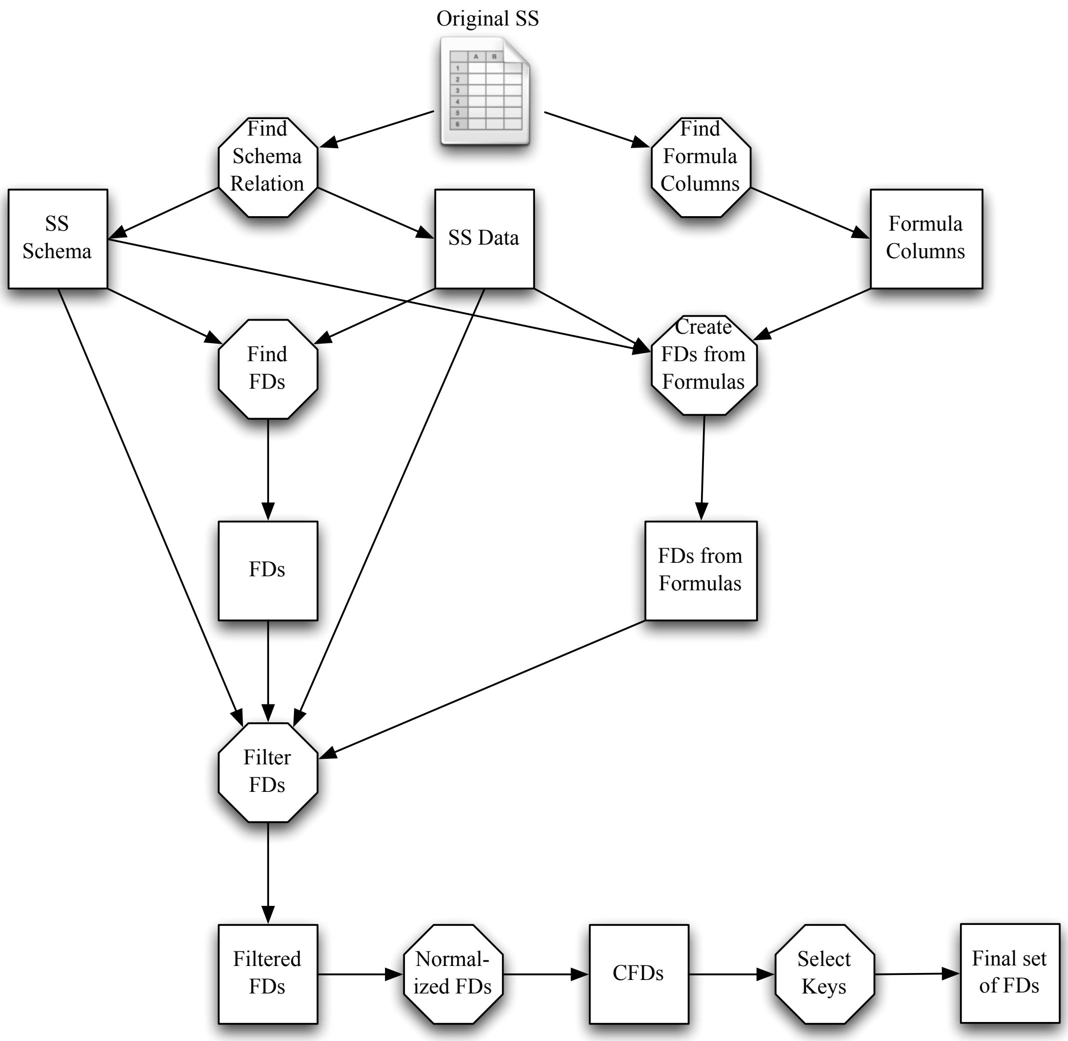


Figure 2.1: Steps necessary to infer functional dependencies from spreadsheets.

We use "regular" data mining techniques to discover all the functional dependencies induced by the spreadsheet data, which usually is a big set. In a database, though, the number of functional dependencies that who designs it have in mind is usually small. This means that the inferred functional dependencies are often polluted by "accidental" dependencies, that is, dependencies that are embedded in the data but do not characterize the relationships between the data. Thus, these dependencies should be discarded.

In this chapter we also describe how to, based on the idiosyncrasies of spreadsheets, select the relevant dependencies and how to discard the ones that do not characterize the data. For example, we take into consideration the order of the columns and their labels to recognize the dependencies that we should keep and the ones that we should discard. In fact, the techniques here presented only apply to spreadsheets because they use characteristics available only in this paradigm.

Having computed a small and very representative set of functional dependencies, does not make it necessarily suitable to represent the spreadsheet data because it may not be normalized. Thus, we present a technique for the normalization of functional dependencies. Only after this step we can use these dependencies, for example, to create a relational database to store the spreadsheet data.

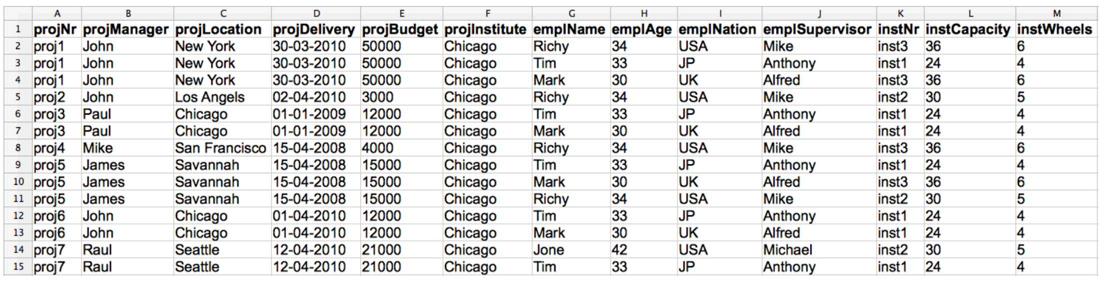
As we will see in Chapter 3, the functional dependencies inferred in this chapter can be used to construct several models specifying the business logic of spreadsheets.

This chapter is organized as follows. In Section 2.2 we present a motivational example used throughout this chapter. In Section 2.3 we introduce some concepts from the relational database realm. In Section 2.4 we explain how to extract functional dependencies from spreadsheets. The extraction of functional dependencies from spreadsheet formulas is presented in Section 2.5. Sections 2.6 and 2.7 describe techniques to filter and normalize functional dependencies for spreadsheets, respectively. In Section 2.8 we explain how to combine all the previous techniques to calculate a set of functional dependencies characterizing a spreadsheet's business logic. Finally, in Section 2.9 we draw our conclusions of this chapter.

2.2 Motivational Example

Throughout this chapter we will use an example adapted from (Alhajj 2003) and illustrated in Figure 2.2.

This spreadsheet reproduces a project management system, gathering information about projects (number, manager, location, delivery date, budget and institute), their workers (name, age, nationality and supervisor) and the instruments they use (number, capacity and number of wheels). The name of each column should give a clear idea



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**projNr**	**projManager**	**projLocation**	**projDelivery**	**projBudget**	**projInstitute**	**emplName**	**emplAge**	**emplNation**	**emplSupervisor**	**instNr**	**instCapacity**	**instWheels**
2	proj1	John	New York	30-03-2010	50000	Chicago	Richy	34	USA	Mike	inst3	36	6
3	proj1	John	New York	30-03-2010	50000	Chicago	Tim	33	JP	Anthony	inst1	24	4
4	proj1	John	New York	30-03-2010	50000	Chicago	Mark	30	UK	Alfred	inst3	36	6
5	proj2	John	Los Angels	02-04-2010	3000	Chicago	Richy	34	USA	Mike	inst2	30	5
6	proj3	Paul	Chicago	01-01-2009	12000	Chicago	Tim	33	JP	Anthony	inst1	24	4
7	proj3	Paul	Chicago	01-01-2009	12000	Chicago	Mark	30	UK	Alfred	inst1	24	4
8	proj4	Mike	San Francisco	15-04-2008	4000	Chicago	Richy	34	USA	Mike	inst3	36	6
9	proj5	James	Savannah	15-04-2008	15000	Chicago	Tim	33	JP	Anthony	inst1	24	4
10	proj5	James	Savannah	15-04-2008	15000	Chicago	Mark	30	UK	Alfred	inst3	36	6
11	proj5	James	Savannah	15-04-2008	15000	Chicago	Richy	34	USA	Mike	inst2	30	5
12	proj6	John	Chicago	01-04-2010	12000	Chicago	Tim	33	JP	Anthony	inst1	24	4
13	proj6	John	Chicago	01-04-2010	12000	Chicago	Mark	30	UK	Alfred	inst1	24	4
14	proj7	Raul	Seattle	12-04-2010	21000	Chicago	Jone	42	USA	Michael	inst2	30	5
15	proj7	Raul	Seattle	12-04-2010	21000	Chicago	Tim	33	JP	Anthony	inst1	24	4

Figure 2.2: A spreadsheet reproducing a project management system.

of the information it represents. The values in column `L` (labeled *instCapacity*) are calculated multiplying the values of column `M` (labeled *instWheels*) by 6.

This spreadsheet defines a valid model to represent the information of the management system. However, it contains redundant information. Notice that, the displayed data specifies seven projects, but their information is included several times. In fact, some of it is repeated three times! This kind of redundancy makes the maintenance and update of the spreadsheet complex and error-prone. A mistake is easily made, for example, by mistyping a name and thus corrupting the data.

The same information can be stored without redundancy. In fact, in the database community, techniques for database normalization are commonly used to minimize duplication of information and improve data integrity. Database normalization is based on the detection and exploitation of *functional dependencies* inherent in the data (Maier 1983; Ullman 1988; Atzeni and De Antonellis 1993).

A *functional dependency* between two columns or sets of columns A and B, denoted $A \rightharpoonup B$, means that when projecting the columns in A, if two rows are equal, then, when projecting the columns in B the corresponding rows are also equal. For instance, the project number functionally determines the project manager.

The following question arises: Can we leverage these database techniques for spreadsheets?

Based on the data in our example spreadsheet, a standard data mining algorithm will infer the following functional dependencies:

$$\begin{aligned}
&projNr \rightharpoonup projManager, projLocation, projDelivery, projBudget, projInstitute\\
&projManager \rightharpoonup projInstitute\\
&projLocation \rightharpoonup projBudget, projInstitute\\
&projDelivery \rightharpoonup projInstitute\\
&projBudget \rightharpoonup projLocation, projInstitute
\end{aligned}$$

$$
\begin{array}{l}
emplName \rightharpoonup projInstitute, emplAge, emplNation, emplSupervisor\\
emplAge \rightharpoonup projInstitute, emplName, emplNation, emplSupervisor\\
emplNation \rightharpoonup projInstitute\\
emplSupervisor \rightharpoonup projInstitute, emplName, emplAge, emplNation\\
instNr \rightharpoonup projInstitute, instCapacity, instWheels\\
instCapacity \rightharpoonup projInstitute, instNr, instWheels\\
instWheels \rightharpoonup projInstitute, instNr, instCapacity\\
projManager, projLocation \rightharpoonup projNr, projDelivery\\
projManager, projDelivery \rightharpoonup projNr, projLocation, projBudget\\
projLocation, projDelivery \rightharpoonup projNr, projManager\\
projManager, projBudget \rightharpoonup projNr, projDelivery\\
projDelivery, projBudget \rightharpoonup projNr, projManager\\
projLocation, emplName \rightharpoonup instNr, instCapacity, instWheels\\
projBudget, emplName \rightharpoonup instNr, instCapacity, instWheels\\
projLocation, emplAge \rightharpoonup instNr, instCapacity, instWheels\\
projBudget, emplAge \rightharpoonup instNr, instCapacity, instWheels\\
projManager, emplNation \rightharpoonup emplName, emplAge, emplSupervisor\\
projLocation, emplNation \rightharpoonup emplName, emplAge, emplSupervisor, instNr,\\
\quad instCapacity, instWheels\\
projDelivery, emplNation \rightharpoonup emplName, emplAge, emplSupervisor\\
projBudget, emplNation \rightharpoonup emplName, emplAge, emplSupervisor, instNr,\\
\quad instCapacity, instWheels\\
projLocation, emplSupervisor \rightharpoonup instNr, instCapacity, instWheels\\
projBudget, emplSupervisor \rightharpoonup instNr, instCapacity, instWheels
\end{array}
$$

Notice that there are 27 functional dependencies embedded in the spreadsheet data! This big number of dependencies makes it very difficult to reason about them because there is too much entropy/noise, that is, too many dependencies that are not useful.

The first problem is that constant columns appear in multiple functional dependencies, for example, column `F`, the institute. This implies that this column will appear in most of the functional dependencies. In fact, it will appear as consequent of any functional dependency that has as antecedent a column with two equal values in different rows. This situation is quite common in spreadsheets and should be handled in a proper way. A possible solution would be to infer the functional dependencies without considering this column and producing a single functional dependency with it: $projInstitute \rightharpoonup \{\,\}$. In this case, the consequent of the dependency is the empty set.

From the set of inferred functional dependencies we would like to filter out the "accidental" dependencies referred in Section 2.1, that is, the dependencies that do not reflect true relationships among the data. We would like to discover the following functional dependencies since they define/represent the four entities involved in a project management system: namely *projects*, *employees*, *instruments* and *institutes*:

$$
\begin{aligned}
&projNr \rightharpoonup projManager, projLocation, projDelivery, projBudget\\
&emplName \rightharpoonup emplAge, emplNation, emplSupervisor\\
&instNr \rightharpoonup instWheels\\
&projInstitute \rightharpoonup \{\,\}
\end{aligned}
$$

The second problem is that spreadsheet formulas can induce functional dependencies too. In our running example, column `L` (labeled *instCapacity*) is calculated using column `M` (labeled *instWheels*). We can say that column `L` is determined by column `M`. So the functional dependency $instWheels \rightharpoonup instCapacity$ exists in our example.

These functional dependencies can be normalized: the content of each dependency can be such that the entire set respects certain properties ensuring data consistency.

A common property in databases is the *lossless decomposition property*: this property ensures that if we decompose a relation/table into smaller relations, it is possible to undo the process and recover the original relation (Maier 1983). To ensure that the set of functional dependencies obeys to this property we can add a particular functional dependency to the current set of dependencies. This dependency has the form $all\_the\_columns\_of\_our\_spreadsheet \rightharpoonup some\_new\_column\_not\_used$. For our running example, such dependency is:

$$
\begin{aligned}
&projNr, projManager, projLocation, projDelivery, projBudget, projInstitute,\\
&\quad emplName, emplAge, emplNation, emplSupervisor, instNr, instCapacity,\\
&\qquad instWheels \rightharpoonup newAtt
\end{aligned}
$$

Normalizing our previous set we get the following new normalized set:

$$
\begin{aligned}
&projNr \rightharpoonup projManager, projLocation, projDelivery, projBudget\\
&emplName \rightharpoonup emplAge, emplNation, emplSupervisor\\
&instNr \rightharpoonup instWheels\\
&projInstitute \rightharpoonup \{\,\}\\
&instWheels \rightharpoonup instCapacity\\
&projNr, emplName, instNr, projInstitute \rightharpoonup \{\,\}
\end{aligned}
$$

In this new set, there are two new functional dependencies: the last one was originated by the dependency introduce to ensure the lossless decomposition property; the one but last was created based in the formula of column `L`.

The results presented in this section were automatically produced by the techniques we will formalize in the following sections. With these dependencies we can construct different models for our spreadsheets. These models can be used to generate edit assistance for end users, migration techniques, evolution and refactoring of spreadsheets. In fact, we will describe these techniques in the following chapters.

2.3 Relational Databases

In this section we briefly introduce some well established concepts from relational database theory. Most of the definitions here presented are taken from (Maier 1983). These definitions are essential for understanding the work we will present in the rest of this thesis.

A *relational schema* R is a finite set of attributes $\{A_1, ..., A_k\}$. Corresponding to each *attribute* A_i is a set D_i called the *domain* of A_i. These domains are arbitrary, non-empty sets, finite or countably infinite. In the context of spreadsheets an attribute usually corresponds to the label of a column, such as *projNr* and *instCapacity*.

A *relation* (or *table*) r on a relational schema R is a finite set of *tuples* (or *rows*) of the form $\{t_1, ..., t_k\}$. For each $t \in r$, $t(A_i)$ must be in D_i. Our example has only one table which corresponds to the entire spreadsheet. In general, a table is a set of rows labeled by a row of attributes or labels.

A *relational database schema* is a collection of relational schemas $\{R_1, ..., R_n\}$. A *relational database* (RDB) is a collection of relations $\{r_1, ..., r_n\}$.

Each tuple is uniquely identified by a minimum non-empty set of attributes called *primary key* (PK).

Some times there may be more than one set of attributes suitable for becoming the primary key. Each of these sets is designated a *candidate key* (CK) and only one is chosen to become the primary key. To represent schemas with candidate keys, we use a pair where its first part represents a set of candidate keys and its second part represents the rest of the attributes. For example,

$$(\{\{projNr\}, \{projNr, projManager\}\}, \{projDelivery, projBudget\})$$

represents the project schema for the running example containing two candidate keys.

A *foreign key* (FK) is a set of attributes within one relation that matches the primary key of some relation.

A *relationship* is a concept from the entity-relationship modeling framework (Chen 1976): a *relationship R* with participant entities (or relations) $R_1, ..., R_n$ defines a set of associations among these entities and is composed by a set of instances. Each instance represents the fact that the entities participating on it are related. A relationship is usually represented by a table in a database.

Some these concepts are illustrated in Figure 2.3.

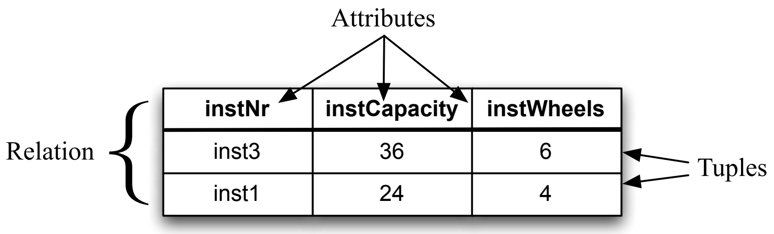


Figure 2.3: An example of a relation representing part of our management spreadsheet.

To represent textually a relational database schema modeling part of our running example, we write:

$Instrument\ (\underline{instNr}, instWheels)$
$< Work >\ (\underline{\#projNr, \#emplName, \#instNr, \#projInstitute})$

The first row represents instruments, so we start by writing the name of the relation/table, *Instrument*. We then write all its attributes within parentheses. The key attributes are underlined. The second row represents the *Work* relationship. It is written in a similar way as *Instrument*, but we need to mark the foreign keys, and thus we add as prefix the symbol # to each attribute that is a foreign key. It is also necessary to show that it is a relationship and so we surround the name with symbols $<$ and $>$.

Given this representation, we now have to different definitions for a relational schema: one where the schema is only a set of attributes, and this new representation with primary and foreign keys and with relationships. When it is unambiguous, we will use the same term for both definitions. When necessary, we will make it clear which one we are using.

A *functional dependency* between two sets of attributes A and B, written $A \rightharpoonup B$, holds in a table if for any two tuples t and t' in that table $t[A] = t'[A] \implies t[B] = t'[B]$ where $t[A]$ yields the (sub)tuple of values for the attributes in A. In other words, if the tuples agree in the values for attribute set A, they agree in the values for attribute set B. The set A is called the *antecedent* of the functional dependency and the set B its *consequent*. The attributes in A can be called the *key* attributes and the ones in B can be called *non-key* attributes.

For instance, in our running example the functional dependency $emplName \rightharpoonup emplAge$ exists, meaning that the values in column *emplName* uniquely determine the values in column *emplAge*, that is, it models the usual notion that an employee cannot have two different names.

Let us take as another example the tables presented in Figure 2.4.

$\mathbf{A_1}$	$\mathbf{A_2}$	$\mathbf{B_1}$	$\mathbf{B_2}$
a_1	a_2	b_1	b_2
a_1	a_3	b_1	b_1
a_1	a_2	b_1	b_2

(a) Table inducing $A \rightharpoonup B$.

$\mathbf{A_1}$	$\mathbf{A_2}$	$\mathbf{B_1}$	$\mathbf{B_2}$
a_1	a_2	b_1	b_2
a_1	a_3	b_1	b_1
a_1	a_2	b_2	b_2

(b) Table not inducing $A \rightharpoonup B$.

Figure 2.4: Two example tables.

Let $A = \{A_1, A_2\}$ and $B = \{B_1, B_2\}$. The data in the table presented in Figure 2.4a encodes the functional dependency $A \rightharpoonup B$ ($\{A_1, A_2\} \rightharpoonup \{B_1, B_2\}$) because rows 1 and 3 have the same values in A and in B. The other row does not affect the dependency because the values in A are different from the others. The data in the table presented in Figure 2.4b does not encode the functional dependency $A \rightharpoonup B$ because, although rows 1 and 3 have the same values in A, the values in B are different. Notice that the value in the third row in column B_1 changed from $b1$ to $b2$ (in red) and thus the functional dependency $A \rightharpoonup B$ does not hold anymore.

Database normalization is important to prevent data redundancy. Although, there are more normal forms, in general, a RDB is considered normalized if it respects the third normal form. Next, we recall the definition of three normal forms (Codd 1972).

The *first normal form* (1NF) is respected if each element of each tuple contains an atomic value. This corresponds to a cell having a single value, for example, a cell with a project number cannot have two project numbers.

A relation respects the *second normal form* (2NF) if it respects the 1NF and its non-key attributes are not functionally dependent on part of the key attributes. For example, for a table with key attributes $\{A, B\}$ and non-key attributes $\{C\}$, C must be functionally dependent on the entire set $\{A, B\}$, that is, not only dependent on A or B.

The *third normal form* (3NF) is respected if the 2NF is present and if the non-key attributes are only dependent on the key attributes. For example, for a table with primary key $\{A\}$ and non-key attributes $\{B,C\}$, it cannot exist the functional dependency $B \rightharpoonup C$ or any other involving the non-key attributes.

2.4 Inferring Functional Dependencies from Spreadsheet Data

In this section we explain how to extract functional dependencies from the data in a spreadsheet. In fact, this process is similar to extract functional dependencies in databases, which is a complex data mining process. There are several data mining algorithms that perform this task in the context of databases such as TANE (Huhtala *et al.* 1999), DEPMINER (Lopes *et al.* 2000), FUN (Novelli and Cicchetti 2001) or FD_MINE (Yao and Hamilton 2007). Since the process is similar in databases and spreadsheets, we adapted an algorithm from databases to work with spreadsheets. Although they produce the same functional dependencies, we choose to use FUN because it is considered to be the faster and the most efficient (Novelli and Cicchetti 2001).

FUN receives as input a set of data tuples, that is, a relation. We consider each row of a spreadsheet as a tuple. Therefore, the columns in a spreadsheet represent the attributes of the spreadsheet. Note that, we could also consider columns as tuples and rows as attributes, but this is a less common way to structure a spreadsheet.

A few concepts, taken from (Novelli and Cicchetti 2001), are necessary to understand this algorithm.

Cardinality The *cardinality* of a set of attributes X in a relation r, written as $|X|_r$, is the number of distinct values of X in r.

Free A set X of attributes is said to be *free* in a relation r if and only if $\nexists X' \subset X, |X'|_r = |X|_r$. Informally, this means that if we project r with the attributes in X we get a certain number of distinct tuples. For X to be said free, there cannot exist a subset of X, say X', such that the projection over r of X' contains the same number of distinct tuples of the projection of X.

Maximal subset A *maximal subset* of X is any of its subsets with one attribute less then itself.

Closure The *closure* of a set X, subset of a schema R, in a relation r is defined as $X_r^+ = X \cup \{A \in R \backslash X \mid |X|_r = |X \cup A|_r\}$. This means that the closure of X contains all the attributes of the relation schema r functionally determined by X.

Quasi-closure The *quasi-closure* of a set X in a relation r is defined as $X_r^\diamond = X \cup \bigcup_{A \in X} (X \backslash A)_r^+$. Informally, the quasi-close of a set of attributes X is the union of the closures of all its maximal subsets and X itself.

FUN is a step-wise algorithm and at each step it handles candidate attributes of increasing length. Each candidate functional dependency is a quadruple with the list of candidate attributes X, its cardinality, the quasi-closure and the closure. At level one it works with candidates of length one computing their cardinality and generating a set of free sets L_0. In the next level all possible pairs of distinct attributes are considered. If the couple in examination is not a free set then it captures at least one functional dependency and at most two minimal functional dependencies that will be yielded by the algorithm. If it is a free set then it is a possible source of functional dependencies and if there is any they will be computed in the next step.

Let us assume that the algorithm is in the step k. If the considered attribute set is proved to be free, it is dealt with at level k. If not, the candidate encompasses at least a maximal subset having a similar cardinality and the target is the additional attribute in the candidate. Each superset of a non-free set is non-free and cannot be source of minimal functional dependency and so it is not examined in the next step. In the kth level all free sets with k attributes are generated as well as all minimal functional dependencies captured by the initial candidates of such a step.

The set of all free sets yielded at each level is used to provide the next level with a set of candidates. Free sets of length k are expanded to give new candidates of length $k+1$ by combining two free sets sharing $k-1$ attributes. This guarantees that only possible candidates are generated. It finishes when no more candidates are generated.

Algorithm 1 shows the steps of FUN. For more detail about the algorithm, the reader is referred to (Novelli and Cicchetti 2001).

We have expressed FUN as the HASKELL *fun* function[1]. The algorithm we are reusing here works with static data, that is, it does not work with the formulas contained in a spreadsheet. The tool we developed evaluates the cells containing formulas

[1]We have defined a bridge between popular spreadsheet systems and the HASKELL programming language so that the spreadsheet data is available. This connection between spreadsheets and HASKELL is part of the HAEXCEL framework and it is described in Chapter 8.

Algorithm 1 Algorithm to infer functional dependencies from data.

```
input: a data set $R$
output: a set of functional dependencies
  $L_0 = < \emptyset, 0, \emptyset, \emptyset >$
  $L_1 = \{< \{A\}, Count(\{A\}), \{A\}, \{A\} > \mid A \in R\}$   /* Count calculates cardinality */
  $R' = R \backslash \{A \mid \{A\}$ is a key$\}$
  for $(k = 1; L_k \neq \emptyset; k = k + 1)$ do
    $ComputeClosure(L_{k-1}, L_k)$                 /* computes closure */
    $ComputeQuasiClosure(L_k, L_{k-1})$            /* computes quasi-closure */
    $DisplayFD(L_{k-1})$                           /* displays a functional dependency */
    $PurePrune(L_k, L_{k-1})$                      /* discards unusable candidates */
    $L_{k+1} = GenerateCandidate(L_k)$             /* generates candidates */
  end for
  $DisplayFD(L_{k-1})$
```

and replaces the formulas by their results. In this way, we get a relation without formulas, as required by the algorithm.

Next, we execute this function in the *ghc* HASKELL interpreter (Jones *et al.* 1993) and show the computed result for the project management system example (the arguments *projectsSchema* and *projectsData* correspond to the first and remaining rows of the spreadsheet of our example, respectively):

```
*ghci> fun [] projectsSchema projectsData
projNr ⇀ projManager, projLocation, projDelivery, projBudget, projInstitute
projManager ⇀ projInstitute
projLocation ⇀ projBudget, projInstitute
projDelivery ⇀ projInstitute
projBudget ⇀ projLocation, projInstitute
emplName ⇀ projInstitute, emplAge, emplNation, emplSupervisor
emplAge ⇀ projInstitute, emplName, emplNation, emplSupervisor
emplNation ⇀ projInstitute
emplSupervisor ⇀ projInstitute, emplName, emplAge, emplNation
instNr ⇀ projInstitute, instCapacity, instWheels
instCapacity ⇀ projInstitute, instNr, instWheels
instWheels ⇀ projInstitute, instNr, instCapacity
projManager, projLocation ⇀ projNr, projDelivery
projManager, projDelivery ⇀ projNr, projLocation, projBudget
```

$projLocation, projDelivery \rightharpoonup projNr, projManager$
$projManager, projBudget \rightharpoonup projNr, projDelivery$
$projDelivery, projBudget \rightharpoonup projNr, projManager$
$projLocation, emplName \rightharpoonup instNr, instCapacity, instWheels$
$projBudget, emplName \rightharpoonup instNr, instCapacity, instWheels$
$projLocation, emplAge \rightharpoonup instNr, instCapacity, instWheels$
$projBudget, emplAge \rightharpoonup instNr, instCapacity, instWheels$
$projManager, emplNation \rightharpoonup emplName, emplAge, emplSupervisor$
$projLocation, emplNation \rightharpoonup emplName, emplAge, emplSupervisor, instNr,$
  $instCapacity, instWheels$
$projDelivery, emplNation \rightharpoonup emplName, emplAge, emplSupervisor$
$projBudget, emplNation \rightharpoonup emplName, emplAge, emplSupervisor, instNr,$
  $instCapacity, instWheels$
$projLocation, emplSupervisor \rightharpoonup instNr, instCapacity, instWheels$
$projBudget, emplSupervisor \rightharpoonup instNr, instCapacity, instWheels$

In fact, our function receives as first argument a list of attributes that it should not consider. This will be useful to handle some attributes in a special way, as it will be explained later on. In this case this list is empty.

The functional dependencies derived by FUN depend on the quantity and quality of the data, and thus, for small samples of data, or data that exhibits too many or too few dependencies, it may not produce the desired functional dependencies. In fact, even the non-natural dependency $emplNation \rightharpoonup projInstitute$ is inferred, implying that the nationality of an employee determines the project institute! Note also that, the *projInstitute* column occurs in most of the dependencies although only one institute appears in that column, namely `Chicago`. Such single value columns are common in spreadsheets. However, for FUN they induce redundant fields and redundant functional dependencies. In Section 2.6 we will explain how to handle these cases.

2.5 Inferring Functional Dependencies from Spreadsheet Formulas

Before we present the inference of dependencies from formulas, let us first define a spreadsheet, and in particular a spreadsheet cell. A *spreadsheet* can be seen as a partial function $S : A \rightarrow V$ mapping addresses to spreadsheet values. An elements of S, that is, a *cell*, is represented as $a = v$. A *cell address a* is taken from the set $A = Letters \ \times \mathbb{N}$, where $Letters = \{A,\ldots,Z,AA,\ldots,AZ,BA,\ldots,BZ,\ldots\}$. A *value* $v \in V$ can be a plain value $c \in C$ like a string or a number, a reference to other cell using addresses, or a formula $f \in F$ that can be applied to one or more values: $v \in V ::= c \mid a \mid f(v,\ldots,v)$.

Spreadsheets use formulas to define the values of some elements in terms of other elements. For example, in the project management system example, the values in column *instCapacity* (column `L`) are computed by multiplying the values in column *instWheels* (column `M`) by 6; this is written as `M2 = L2 × 6` (for the second row). This formula states that the values in column `M` determine the values in column `L`, thus inducing the following functional dependency: *instWheels* $\rightharpoonup$ *instCapacity*.

Since the values of the antecedent of a functional dependency cannot change over time, we do not consider columns defined by formulas as candidates to antecedents when inferring dependencies.

In general, formulas can have references to other cells also containing formulas. Consider, as an example, the formula `A1 = B1 * C1` in column `A`, and the formula `C1 = D1 + E1` in column `C`. Notice that, these formulas are for the first row of the spreadsheet and that their row reference increments naturally throughout the rest of the rows. Because `C` is defined by another formula, the values that determine `C` also determine `A`. As a result, the two formulas induce the following functional dependencies:

$$B,C \rightharpoonup A$$
$$D,E \rightharpoonup C$$

Expanding the relationship between formulas, we get the following result:

$$B,D,E \rightharpoonup A$$
$$D,E \rightharpoonup C$$

The function *forms2fds*, presented below, receives the formulas in a spreadsheet and returns all the functional dependencies induced by them. This function uses an auxiliary one: *getCols*. Suppose K and L are columns, M is a row and c a plain value. The function *getCols* returns the set of columns used in a formula.

$$getCols : \{Formula\} \rightarrow Formula \rightarrow \{CellAddress\}$$
$$getCols\ all\ c = \{\,\}$$
$$getCols\ all\ KM = getCols\ all\ (lookup\ KM\ all)$$
$$getCols\ (f\ (K1,\ldots,KM)) = getCols\ all\ K1\ \cup \ldots \cup\ getCols\ all\ KM$$

If the cell is defined by a constant (*c*), it returns nothing; if the cell is a reference to another cell (*KM*), it will search for the definition of such cell (*lookup KM all*) and recursively applies itself to this result. If it is defined by a function, it recursively applies itself to the arguments of this function returning the union of these results.

The function *forms2fds* receives a set of formulas and returns all the functional dependencies induced by them.

$$
\begin{aligned}
&\mathit{forms2fds} : \{\mathit{Formula}\} \to \{\mathit{Formula}\} \to \{\mathit{FD}\}\\
&\mathit{forms2fds}\ \mathit{all}\ (\{\mathit{LN} = c\} \cup \mathit{rs}) = \mathit{forms2fds}\ \mathit{all}\ \mathit{rs}\\
&\mathit{forms2fds}\ \mathit{all}\ (\{\mathit{LN} = \mathit{KM}\} \cup \mathit{rs}) = \{\mathit{getCols}\ \mathit{all}\ \mathit{KM} \rightharpoonup \mathit{LN}\} \cup \mathit{forms2fds}\ \mathit{all}\ \mathit{rs}\\
&\mathit{forms2fds}\ \mathit{all}\ (\{\mathit{LN} = f\} \cup \mathit{rs}) = \{\mathit{getCols}\ \mathit{all} f \rightharpoonup \mathit{LN}\} \cup \mathit{forms2fds}\ \mathit{rs}
\end{aligned}
$$

In the case a formula is a plain value (first alternative in the definition of function *forms2fds*), the function recursively gets the dependencies in the remaining cells *rs*. In the second case, the cell contains a reference to cell *KM*, and so, the function will use *getCols* to get all the cells referenced by *KM*. This result is then used as the antecedent of a functional dependency, whilst its consequent will be the *LN* cell. The function will then try to find the rest of the functional dependencies. In the case the cell is defined by a function, it dereferences all references in the formula and creates from that an antecedent. As expected, the consequent of the resulting functional dependency is the cell *LN*. The function then continues working on the remaining cells *rs*.

2.6 Filtering Functional Dependencies

As we explained before, there are some functional dependencies that are inferred from the data, but they do not reflect any valid relationship in the data. For example, the dependency $\mathit{emplNation} \rightharpoonup \mathit{projInstitute}$ is induced by the data in the spreadsheet example illustrated in Figure 2.2. This dependency implies that the nationality of an employee determines the project's institute! Therefore, dependencies like this should be discarded of further reasoning. In fact, we want the functional dependencies that better characterize the business logic of the underlying spreadsheet.

The amount of "accidental" functional dependencies heavily depends on the data and how well the relationships among it are represented. Usually, spreadsheets with little data induce more "accidental" functional dependencies.

The knowledge about the functional dependencies in a spreadsheet provides the basis to identifying relationships among the data. In fact, they are the building blocks for the models we will produce in the following chapters. The more accurate we can make this filtering step, the better the inferred models will reflect the actual business logic of the spreadsheets.

The process of identifying the "valid" functional dependencies is, of course, ambiguous in general. Therefore, we employ a series of heuristics to evaluate dependencies. Each of them can add support to a functional dependency. Note that, the smaller the number attributed to the support the better (this made the implementation of the heuristics simpler). In the following paragraphs we describe the employed heuristics.

Label semantics This heuristic is used to classify antecedents in functional dependencies. Most keys in databases are labeled as "code" or "number" or are a combination of these labels with a label more related to the subject. For example, in our projects spreadsheet, the label *projNr* is used to label the column with the projects unique identifiers. Therefore, we consider labels "id", "code", "number", "nr", "no" and combinations of them with other labels. A functional dependency with an antecedent attribute of this kind receives higher support than the others. Each attribute considered as a possible key decreases or increases the support number of the functional dependency in one value, depending if it is on the antecedent or consequent of the functional dependency, respectively.

Label arrangement It is common to have columns with unique identifiers before the rest of the data related to some entity. Take as an example the instrument entity of the projects spreadsheet: the instrument unique number appears before its capacity and number of wheels. This can be indicative of that column being a key. As expected the algorithm gives more support to the functional dependencies that respect the order of the columns in the spreadsheet. In our running example, the functional dependency *instNr* $\rightharpoonup$ *instCapacity* has more priority than the functional dependency *instCapacity* $\rightharpoonup$ *instNr*. The support for each functional dependency is calculated as follows: imagine that we arrange all the attributes of the functional dependency in a list starting with the antecedent attributes and then the consequent ones. Starting from the end of this list, we verify if the last element has a bigger index than the one before in the schema under consideration. If so, we increase the support number of this functional dependency in one unit. We do the same for all the attributes in the list.

Distance between labels It is reasonable to think that a spreadsheet user adds columns in such a way that related things are close to each other. It is also reasonable to think that the further apart the columns are, the less related they are. Based on these principles we calculate the distance between the attributes of a functional dependency and give more priority to those dependencies that have a smaller distance. To calculate this support number, we calculate the indexes of the attributes in the functional dependency in the schema and create a sorted list with them. If this indexes are all consecutive, the functional dependency receives support number 0. Otherwise, the difference between each two consecutive indexes is calculated and added to the support number.

After sorting we subtract the index of the first index to the second, the second to the third, and so on, until the end of the list. If the result is negative, we return it as a positive number and add 10 to it. These numbers have been used in several examples and have produced good results.

Antecedent size Good keys often consist of a small number of attributes (Connolly and Begg 2001). In fact, most keys in databases are composed by a single attribute. Therefore, the smaller the number of antecedent attributes, the higher the support for a functional dependency. The support number for a functional dependency is the length of its antecedent: the bigger it is, the less support it will have.

Ratio between antecedent and consequent sizes In general, functional dependencies with smaller antecedents and bigger consequents are stronger and thus more likely to be a reflection of the underlying data model. Therefore, a functional dependency receives a higher support, the smaller the ratio between the number of consequent attributes and the number of antecedent attributes is. The support number is calculated dividing the length of the antecedent by the length of the consequent.

Note that several of these heuristics are possible only in the context of spreadsheets. This observation supports the contention that end-user software engineering can benefit greatly from the context information that is available in a specific end-user programming domain. In the spreadsheet domain, rich context is provided through the spatial arrangement of cells and through labels (Erwig 2009).

After the algorithm has calculated these supports for the functional dependencies, they are all summed and the dependencies sorted according to their support. The algorithm then selects the functional dependencies from that list in the order of their support until all the attributes of the schema are at least in one functional dependency.

Based on these heuristics, our algorithm produces the following dependencies for the projects application:

$$
\begin{array}{l}
projNr \rightharpoonup projManager, projLocation, projDelivery, projBudget \\
emplName \rightharpoonup emplAge, emplNation, emplSupervisor \\
instNr \rightharpoonup instWheels \\
projInstitute \rightharpoonup \{\,\} \\
instWheels \rightharpoonup instCapacity
\end{array}
$$

This result contains five functional dependencies representing the entities one would expect to see in project system. Projects, employees, instruments and their characteristics and institutes are all represented. In fact, what is missing is the relationships between all of them, which will be handled in the next section.

In the next section we explain how to normalize a set of functional dependencies. This normalization step is crucial to avoid insertion, modification and deletion anomalies.

2.7 Normalizing Functional Dependencies

One of the purposes of having data in databases is to avoid some problems like data inconsistence or corruption. Unfortunately, a database *per se* is not enough to guarantee this: it is necessary that the database is normalized. Although there are more normal forms, the *third normal form* (3NF) is usually enough to guarantee that a database is well structured. In particular this avoids three major problems: *insertion*, *modification* and *deletion anomalies* (Codd 1972). The first problem occurs when we need to insert data, but not a complete row (in the context of spreadsheets, for example). Suppose we want to add a new employee to our projects spreadsheet. If he is supposed to work in some existing project, we need to insert again the data about the project. If a mistake is made, data will become corrupted. If we were working with a normalized database, there would exist a table for employees, and thus, there would be no need to reinsert the project's data. The second problem occurs when we change information in one row but leave the same information unchanged in the others or modify it in a non-consistent way. In our example, this may happen if we change the number of wheels of instrument `inst3` in row 2 and do not update the others in the exactly same way. In a normalized database that value occurs only once in the instrument table and so the problem will never occur. The third problem happens when we delete some row and lose other information as a side effect. For example, if we delete row 14 in the projects spreadsheet all the information about the employee `Jone` will be lost. In a normalized database, this will be avoided since each entity will have in its own table.

Maier developed an algorithm, SYNTHESIZE , that normalizes a set of functional dependencies (Maier 1983). It receives a set of functional dependencies as argument and returns a relational database schema respecting the 3NF. In fact, it returns a set of compound functional dependencies: a *compound functional dependency* (CFD) has the form $(X_1,\ldots,X_n) \rightharpoonup Y$, where $X_1,\ldots,X_n$ are all distinct sets of attributes and Y is also a set of attributes. A relation r holds the CFD $(X_1,\ldots,X_n) \rightharpoonup Y$ if it holds the functional dependencies $X_i \rightharpoonup X_j$ and $X_i \rightharpoonup Y$, where $1 \leqslant i$ and $j \leqslant k$. In this CFD $(X_1,\ldots,X_n)$ is the *left side*, $X_1,\ldots,X_n$ are the *left sets* and Y is the *right side*. In fact, a compound functional dependency is nothing more than a shorthand way of writing a set of functional dependencies with equivalent antecedents (Maier 1983).

Algorithm 2 describes SYNTHESIZE.

Algorithm 2 Algorithm to normalize a set of functional dependencies up to the 3NF.

```
input: a set of functional dependencies $F$
output: a set of compound functional dependencies
  $G$ = a reduced, minimum annular cover for $F$
  for all CFD $(X_1,X_2,...,X_n) \rightharpoonup Y$ in $G$ do
    $G' = G' \cup \{$relational schema $R = X_1X_2...X_nY$ with CKs $\{X_1,X_2,...,X_n\}\}$
  end for
  return the set of relational schemas $G'$
```

The algorithm works as follows: it finds a reduced, minimum annular cover G for the input set of dependencies F. Then, for each compound functional dependency $(X_1,X_2,...,X_n) \rightharpoonup Y$ in the G set, it creates a relational schema with attributes $\{X_1,X_2,...,X_n,Y\}$ and candidate keys $\{X_1,X_2,...,X_n\}$. More detail about this algorithm can be found in (Maier 1983).

We have implemented this algorithm in HASKELL as the *synthesize* function. It receives as argument a set of functional dependencies (list of functional dependencies in HASKELL) and returns a set of compound functional dependencies. Next, we execute *synthesize* with the functional dependencies induced by our running example and show its result:

$(\{\{projNr\}\},\{projManager,projLocation,projDelivery,projBudget\})$
$(\{\{emplName\}\},\{emplAge,emplNation,emplSupervisor\})$
$(\{\{instNr\}\},\{instWheels\})$
$(\{\{instWheels\}\},\{instCapacity\})$
$(\{\{projInstitute\}\},\{\})$

An important property to guarantee in a set of relational schemas is the *lossless decomposition* property. *Lossless decomposition* means that if we decompose a relation into smaller relations, it is possible to undo the process and recover the original relation.

Moreover, it is possible that some columns of our spreadsheet are not included in any functional dependency.

To guarantee that they appear in the final schema and to ensure the lossless decomposition property we need to give an extra functional dependency to the SYNTHESIZE algorithm. In fact, we use Maier's strategy (Maier 1983): we generate an additional functional dependency which contains all the columns of our spreadsheet as antecedent (the exception are the ones defined by formulas). The consequent of the dependency is a newly introduced attribute and the attributes representing formulas. For our example such functional dependency is as follows:

$$\begin{array}{l} projNr,projManager,projLocation,projDelivery,projBudget,emplName, \\ \quad emplAge,emplNation,emplSupervisor,instNr,instWheels,projInstitute \\ \qquad \rightharpoonup instCapacity,newAttribute \end{array}$$

The final set of schemas in the 3NF is as follows:

$(\{\{projNr\}\},\{projManager,projLocation,projDelivery,projBudget\})$
$(\{\{emplName\}\},\{emplAge,emplNation,emplSupervisor\})$
$(\{\{instNr\}\},\{instWheels\})$
$(\{\{instWheels\}\},\{instCapacity\})$
$(\{\{projInstitute\}\},\{\})$
$(\{\{projNr,emplName,instNr,projInstitute\}\},\{\})$

In the next section, we will explain how to combine the techniques introduced in the previous sections to compute functional dependencies from a spreadsheet.

2.8 SSFUN: Functional Dependencies for Spreadsheets

In the previous sections we described how to infer functional dependencies from spreadsheet data, how to filter these dependencies and how to normalize them. In this section, we explain how to compose all these techniques to calculate a set of functional dependencies that completely characterize a spreadsheet, as shown in Algorithm 3.

Algorithm 3 Algorithm to calculate functional dependencies for spreadsheets.

```
input: a spreadsheet $s$
output: a set of sets of functional dependencies
  $schemasAndRelations = findSchemaAndRelation\ s$
  for all $(schema, relation)$ in $schemasAndRelations$ do
    $onesAtts = get1col\ (schema, relation)$
    $oneFDs = map\ (\lambda att \rightarrow (att, \{\ \}))\ onesAtts$
    $formulaAtts = findFormulaCols\ schema\ relation$
    $formulaFDs = createFormulaFDs\ formulaAtts\ schema\ relation$
    $f = fun\ (onesAtts \cup formulaAtts)\ schema\ relation$
    $g = filter\ (f \cup onesFDs \cup formulaFDs)\ schema\ relation$
    $h = synthesize\ relation\ (lossless?)\ g$
    $j = selectKeys\ h$
    $k = toFDs\ j$
  end for
  return set of functional dependencies $k$
```

The first step of our algorithm is to find the schemas and relations of the spreadsheet. To perform this step we use an algorithm from the *UCheck* project (Abraham and Erwig 2007b) that can infer tables in spreadsheets. For each table in the spreadsheet, the algorithm assumes that the first row contains the labels and the remaining rows contain the data. If references exist from one table to another, the functional dependencies are handled as a unique set since these references lead to dependencies between the two schemas. Otherwise, each table is handled on its own. The next steps are executed for each schema and relation.

As we explained before, columns that have the same value in all rows are problematic and should be handled properly. To do so, the algorithm finds these columns, and for each one it produces a functional dependency where the its label is the antecedent. The consequent is the empty set. These columns are also argument of *fun* so it does not try to find functional dependencies containing them.

The next phase is to find the columns that are defined by formulas. For each column, the algorithm verifies if the second row (remember that the first one contains the column label) is defined by a formula. If this is the case, it verifies the remaining rows and collects all the columns of the references that are used to define the formula. These columns are then passed as argument to the function *fun* so it does not search for dependencies with antecedents on this list. Next, the algorithm constructs the functional dependencies based on these columns and their formulas.

The algorithm then runs *fun* with formula and single value columns, the schema and the relation as arguments. The result set of functional dependencies f is then concatenated with the dependencies constructed based on the columns defined by formulas. The result functional dependencies are the argument of *synthesize*. It will also receive a flag indicating if it should normalize respecting the lossless decomposition property or not.

The next step is to select the keys from the candidate keys produced by *synthesize*. The algorithm chooses the smallest candidate key to become the key of the schema since good keys are usually small. This set of schemas is then transformed into a set of functional dependencies: each schema of the form $(key, attributes)$ results in the functional dependency $key \rightharpoonup attributes$. This set of functional dependencies is the result of our algorithm.

For our project management system, the result set of functional dependencies is listed next:

$$\begin{array}{l}
projNr \rightharpoonup projManager, projLocation, projDelivery, projBudget \\
emplName \rightharpoonup emplAge, emplNation, emplSupervisor \\
instNr \rightharpoonup instWheels \\
instWheels \rightharpoonup instCapacity \\
projInstitute \rightharpoonup \{\,\} \\
projNr, emplName, instNr, projInstitute \rightharpoonup \{\,\}
\end{array}$$

Notice that, each of the schemas produced by *synthesize* has a single candidate key which makes the choice for a key obvious. This happens because the set of dependencies that is given to the *synthesize* function is small and thus less likely to be not normalized. Nevertheless, we must guarantee the normalization of the set of dependencies, and thus, we must always run SYNTHESIZE.

2.9 Conclusions

In this chapter we have described how to infer, filter and normalize functional dependencies for spreadsheets. In fact, we have described how to calculate a set of functional dependencies that completely characterize a spreadsheet business logic.

The first step is to infer the functional dependencies. We have reused an existing and efficient algorithm, FUN, developed in the context of databases. However, it produces too many functional dependencies. Algorithms like this produce all the dependencies existing in a data set, but many of them are unexpected. For our running example, there exists the dependency *emplNation* $\rightharpoonup$ *projInstitute* that implies that the nationality of an employee determines the project's institute, which does not make sense. This kind of dependencies has to be discarded.

So, the next logic step is to filter the dependencies that do not reflect any valid relationship among the spreadsheet data. Using spreadsheets idiosyncrasies like the order of columns or the meaning of labels we can calculate a set of dependencies that describe the relationships between columns of a spreadsheet. We also used other heuristics, such as the number of attributes both in the antecedent and in the consequent of dependencies, to produce a set of functional dependencies that more precisely define the business logic of the spreadsheet.

Finally, these functional dependencies must be normalized so they can be useful. Otherwise, the models produced from them may allow corrupted data. In this step we reused an algorithm from Maeir that produces a set of schemas normalized up to the third normal form. We then transform these schemas into a set of normalized functional dependencies. These dependencies can then be used to several purposes, namely the creation of several models representing the spreadsheet business logic.

In the following chapters we will show the different models we can compute using these dependencies and, further on, the possibilities that these models bring to spreadsheet end users.

Chapter 3

Inferring Models for Spreadsheets

Summary

Many errors in spreadsheet formulas can be avoided if spreadsheets are built automatically from higher-level models that can encode and enforce consistency constraints. However, designing such models is time consuming and requires expertise beyond the knowledge to work with spreadsheets.
To address these problems and to support the model-driven spreadsheet engineering approach, we have developed a technique to automatically infer relational database schemas, ClassSheet models and UML class diagrams from spreadsheets. We have integrated our techniques with the HAEXCEL *framework, which allows the automatic generation of refactored spreadsheets from the inferred ClassSheet model. The resulting spreadsheet guides further changes and safeguards the spreadsheet against a large class of formula errors.*

3.1 Introduction

In recent years the spreadsheet research community has recognized the need to support *end-user model-driven software development*, and to provide spreadsheet developers and end users with methodologies, techniques and the necessary tool support to improve their productivity: in fact, several techniques have been proposed to allow end users to safely edit spreadsheets, like, for example, the use of spreadsheet templates (Abraham *et al.* 2005) and *ClassSheets* (Engels and Erwig 2005). All these approaches aim at a form of model-driven software development: a spreadsheet "business model" is defined from which a customized spreadsheet application is generated to guarantee the consistency of the spreadsheet with the underlying model.

Despite of its benefits, model-driven software development is sometimes difficult to realize in practice. For example, in the context of spreadsheets, the use of model-driven software development requires that the developer is familiar both with the spreadsheet domain and with model-driven software development. As some studies suggest, defining the business model of a spreadsheet can be a complex task for end users (Abraham and Erwig 2006a). As a result, end users are unable (or reluctant) to follow this spreadsheet development discipline. Things get even more complex when end users need to modify a large (legacy) spreadsheet developed by others and whose functionality they do not understand.

Legacy spreadsheets pose a particular challenge to the approach of controlling spreadsheet evolution through higher-level models, because the need for a model might be overshadowed by two problems:

1. there are no large short-term benefits, since the spreadsheet is already created, but only smaller longer term benefits, when making future changes to the spreadsheets;
2. the existing data must be transferred into the new model-generated spreadsheet.

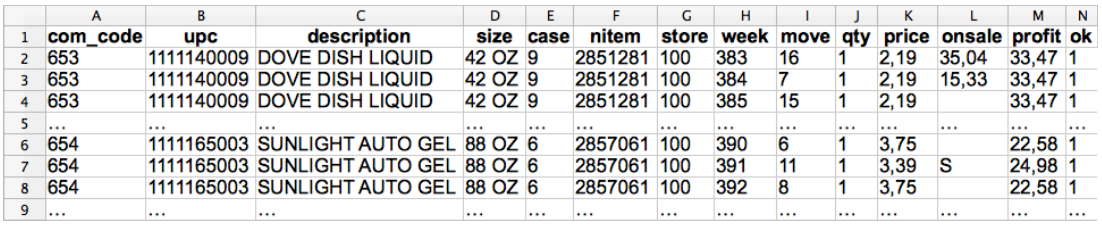
In this chapter, we propose reverse engineering techniques to derive relational database schemas, *ClassSheet* models and UML class diagrams from existing spreadsheets. The algorithm to compute functional dependencies from spreadsheet data introduced in Chapter 2 is the main building block to construct a relational database schema. The induced functional dependencies and this schema are then used to produce a *ClassSheet* model representing the business logic of the spreadsheet. Since *ClassSheet* models are similar to UML class diagrams we also introduce a transformation from the former to the latter.

We build three different model because they have different purposes: the relational model is necessary to support the migration to databases, as it will be presented in Chapter 5; the *ClassSheet* is a more specific model for spreadsheets and thus better to represent spreadsheet evolution as we will expose in Chapter 6; finally, the UML model is more generic and well know among different communities supporting migrations between different paradigms. Although we do not make a specific use of this model in this thesis, this gives support for further migrations of spreadsheets.

This chapter is organized as follows. In Section 3.2 we present a motivational example. In Section 3.3 we present the technique to construct a relational database schema from functional dependencies. The algorithm for the automatic derivation of *ClassSheets* for spreadsheets is explained in Section 3.4. The transformation of *ClassSheets* into UML class diagrams is exposed in Section 3.5. In Section 3.6 we present an evaluation of our techniques and Section 3.7 concludes the chapter.

3.2 Motivational Example

Consider the example spreadsheet illustrated in Figure 3.1, adapted from (Powell and Baker 2003).



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	com_code	upc	description	size	case	nitem	store	week	move	qty	price	onsale	profit	ok
2	653	1111140009	DOVE DISH LIQUID	42 OZ	9	2851281	100	383	16	1	2,19	35,04	33,47	1
3	653	1111140009	DOVE DISH LIQUID	42 OZ	9	2851281	100	384	7	1	2,19	15,33	33,47	1
4	653	1111140009	DOVE DISH LIQUID	42 OZ	9	2851281	100	385	15	1	2,19		33,47	1
5	...	...	...	...	...	...	...	...	...	...	...	...	...	...
6	654	1111165003	SUNLIGHT AUTO GEL	88 OZ	6	2857061	100	390	6	1	3,75		22,58	1
7	654	1111165003	SUNLIGHT AUTO GEL	88 OZ	6	2857061	100	391	11	1	3,39	S	24,98	1
8	654	1111165003	SUNLIGHT AUTO GEL	88 OZ	6	2857061	100	392	8	1	3,75		22,58	1
9	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Figure 3.1: A spreadsheet representing a sales system for dishwasher detergents.

This spreadsheet represents a sales system for dishwasher detergents. The labels have the following meanings: **com_code** stands for the commercial code of the detergent, **upc** for universal product code, **description** for the description of the detergent, **size** for the size of the detergent's case, **case** for number of cases, **nitem** for the item number, **store** for the store where the sale took place, **week** for the week the sale was done, **move** for sales quantity, **qty** for the transaction quantity (always set to 1), **price** for the price of each unit, **onsale** for an indicator of advertised sale, **profit** for the store profit per case and **ok** for a confirmation code. Notice that **profit** is calculated using the formula that multiplies the sale's quantity (column `I`) by the price (column `K`) and taking 20% of this: `=I2*K2*0.2` (for the first row).

The business logic underlying this spreadsheet is not immediately clear and it is quite difficult to infer for a non-expert in this domain. In this section, we will informally describe a strategy to infer such a business logic from the spreadsheet data.

Entities contained in such a spreadsheet and relationships between them are reflected by the presence of functional dependencies between spreadsheet columns. For example, using the techniques presented in Chapter 2 we can infer the following functional dependencies:

$$
\begin{aligned}
&ok \rightharpoonup \{\,\}\\
&qty \rightharpoonup \{\,\}\\
&move, price \rightharpoonup profit\\
&store, week \rightharpoonup onsale\\
&upc \rightharpoonup com\_code, description, size, case, nitem\\
&ok, qty, move, week, upc, store \rightharpoonup \{\,\}
\end{aligned}
$$

From these functional dependencies, we would like to generate the following relational schema:

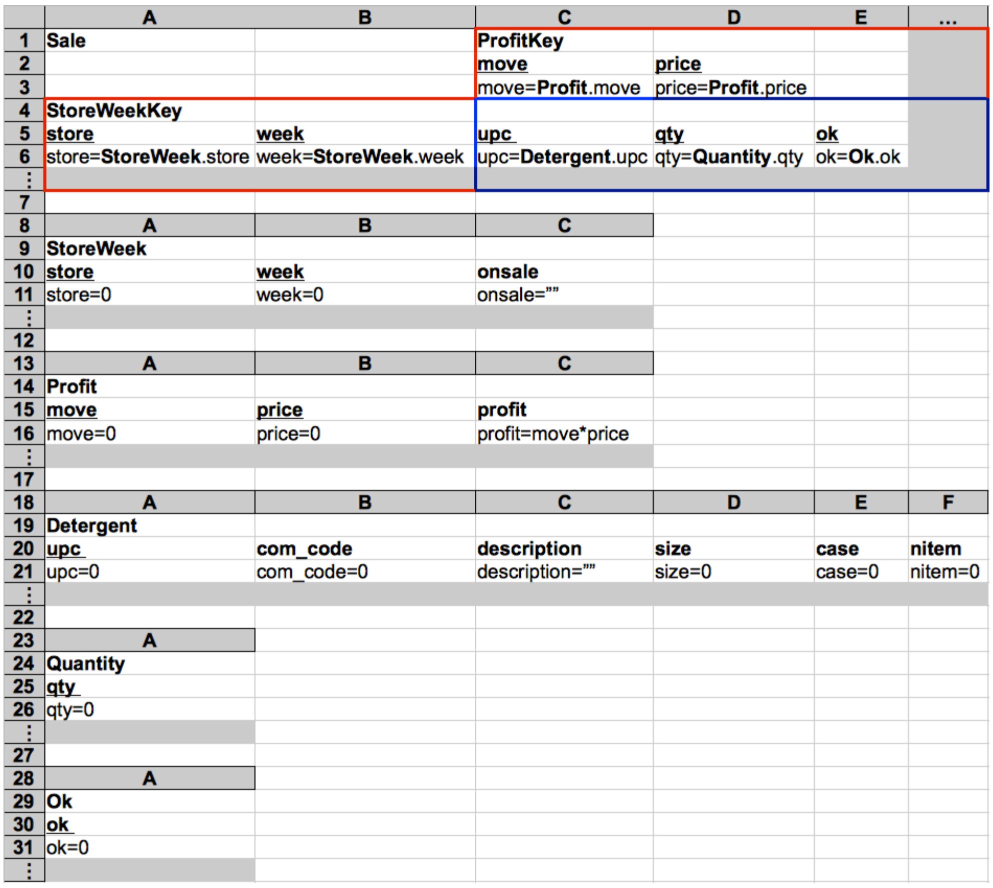
$$
\begin{array}{ll}
Ok & (\underline{ok})\\
Quantity & (\underline{qty})\\
Profit & (\underline{move, price}, profit)\\
StoreWeek & (\underline{store, week}, onsale)\\
Detergent & (\underline{upc}, com\_code, description, size, case, nitem)\\
< Sale > & (\underline{\#upc, \#move, \#price, \#store, \#week, \#qty, \#ok})
\end{array}
$$

The model has several relations: the *Profit* relation stores information about the profit based on the movements and prices; *StoreWeek* contains information that varies for each store every week such as the advertised products; *Detergent* contains all the information about the detergents. *Sale* is a relationship that stores the information on the sales themselves. This relationship is also responsible for guaranteeing the lossless decomposition property. Finally, the relation *Ok* and *Quantity* contain a single value (1 in both cases) which is then referenced from the *Sale* relation.

Although this relational model is very expressive, it cannot completely specify spreadsheets since they need to have a layout specification. In contrast, the *ClassSheet* modeling framework offers high-level, object-oriented formal models to specify spreadsheets and thus presents a promising alternative (Engels and Erwig 2005). *ClassSheets* allow users to express business object structures within a spreadsheet using concepts from UML. A spreadsheet application consistent with the model can be automatically generated, and thus, a large variety of errors can be prevented.

We therefore employ *ClassSheets* as the underlying modeling approach for spreadsheets and transform the inferred relational model into a *ClassSheet* model. Notice that all the information represented in the relational model can be represented in the

ClassSheet. In Figure 3.2 we present the *ClassSheet* specifying our running example.



	A	B	C	D	E	...
1	Sale		ProfitKey			
2			move	price		
3			move=**Profit**.move	price=**Profit**.price		
4	StoreWeekKey					
5	store	week	upc	qty	ok	
6	store=**StoreWeek**.store	week=**StoreWeek**.week	upc=**Detergent**.upc	qty=**Quantity**.qty	ok=**Ok**.ok	
⋮						
7						
8	A	B	C			
9	StoreWeek					
10	store	week	onsale			
11	store=0	week=0	onsale=""			
⋮						
12						
13	A	B	C			
14	Profit					
15	move	price	profit			
16	move=0	price=0	profit=move*price			
⋮						
17						
18	A	B	C	D	E	F
19	Detergent					
20	upc	com_code	description	size	case	nitem
21	upc=0	com_code=0	description=""	size=0	case=0	nitem=0
⋮						
22						
23	A					
24	Quantity					
25	qty					
26	qty=0					
⋮						
27						
28	A					
29	Ok					
30	ok					
31	ok=0					
⋮						

Figure 3.2: *ClassSheet* modeling the sales system for dishwasher detergents.

To present *ClassSheets* we start by considering the simplest table in our example: between rows 28 and 32 (row 32 is the one after row 31 with the ellipsis instead of the row number) is represented the entity to store the *Ok* values. This kind of entity is designated a *class* and, in some cases, represents a table from the relational model. The *Ok* class is represented by the column in which it will be in a spreadsheet, `A` in this case, the label of the table, **Ok**, the label of the columns, in this case just one, **ok**, and the default value for each new row, 0 in this case. If the column represents a key in the relational model, it is underlined, as is the case of *ok*. Notice that, what should be row 32 has ellipses meaning that this column can be vertically expandable, that is, new rows can be added. The *Quantity* class is quite similar to *Ok*.

The *Detergent* table/class is specified in a similar way, but in this case it is composed by six columns.

The top part (rows 1 to 6 plus the following row) represents the sales relation. Since *Sale* is a relationship between two other relations, namely *Profit* and *StoreWeek* it is represented as a *cell class* (blue table) and the two related classes *Profit* and *StoreWeek* in red. This will be explained in detail in the coming sections.

ClassSheets carry information rich enough to allow the automatic generation of UML class diagrams. Figure 3.3 shows the UML class diagram that can be derived from this *ClassSheet*.

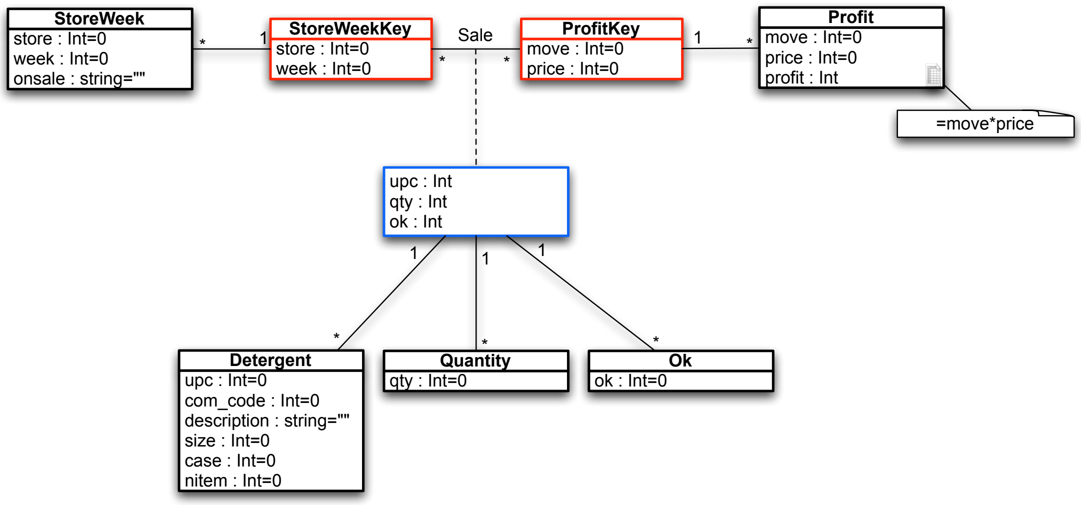


Figure 3.3: A class diagram specifying the sales system for dishwasher detergents.

Although the business logic of the spreadsheet can be represented in the UML model, its layout is lost since it is not possible to represent it using this language. The major implication is that, if a spreadsheet is generated from one of these specifications, a new layout must be created.

In this section, we have described our approach to derive relational schemas, *Class-Sheets*, and UML class diagrams for spreadsheets through an example. In fact, these models were automatically computed by the tool that we have implemented based on our approach. In the coming sections we will describe these steps in a general way for any spreadsheet.

3.3 Deriving a Relational Schema

In this section we explain how to use functional dependencies extracted from spreadsheet data, as exposed in Chapter 2, to compute a complete relational schema. Algorithm 4 shows the main steps of this process.

Algorithm 4 Algorithm to calculate a relational schema from schemas with candidate keys.

```
input: a set of schemas with candidate keys
output: a complete relational schema
1: Infer relations' names
2: Classify candidates keys
3: Infer foreign keys
4: Infer primary keys
5: Create a relational intermediate direct graph
6: Optimize the relational intermediate direct graph
7: Map the relational intermediate direct graph to a relational schema
```

Algorithm 4 is illustrated in a graphical way in Figure 3.4.

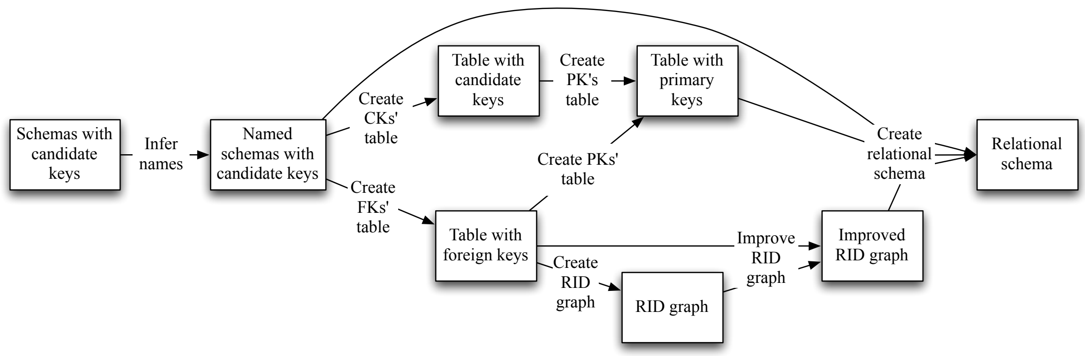


Figure 3.4: Graphical representation of Algorithm 4.

Algorithm 4 receives a set of schemas with candidate keys, but the final result computed in Chapter 2 is a set of functional dependencies, so we cannot use it directly. In fact, we must use an intermediate result containing the required set of schemas containing candidate keys. This result is the output of the SYNTHESIZE algorithm presented in Section 2.7. We show next the schema with candidate keys for our running example.

$(\{\{ok\}\},\{\})$
$(\{\{qty\}\},\{\})$
$(\{\{move,price\}\},\{profit\})$
$(\{\{store,week\}\},\{onsale\})$
$(\{\{upc\}\},\{com\_code,description,size,case,nitem\})$
$(\{\{ok,qty,move,week,upc,store\}\},\{\})$

This set of attributes with candidate keys will be used throughout this section to compute a relational schema. This algorithm has seven steps; each of them is presented in detail in the next sections. Note that, the steps 2 through 6 are based on the algorithm presented by Alhajj (2003). Since it was designed in the context of databases, we had to adapt it to better work in the spreadsheet realm.

We present next the first step of our algorithm, that is, the inference of names for relations.

3.3.1 Name Inference

To generate models and subsequent spreadsheets similar to the ones a human would produce, we perform name analysis on the schema attributes. To give the correct name to each part of the model should improve its understandability.

Let us take as an example the functional dependency *clientNr* $\rightharpoonup$ *clientNm*. One could say that it represents something about *client*, since both column names have "client" as prefix.

To infer names for relations, we apply several heuristics, as described next:

Usual names We have a set of words that are usually used to name databases, tables, keys and attributes. These words are divided in several categories:

Schemas/databases "dvds", "houses", "clients", "customers", etc.

Relations/tables "client", "customer", "house", "dvd", "artist", etc.

Relationships "sale", "enrollment", "works", "studies", etc.

Primary keys "id", "code", "number", a noun concatenated with the previous words, etc.

Attributes "name", "address", "age", "duration", etc.

These names are organized in a hierarchical way, for example, if the name of the database is "dvds", the relations' names are reduced to the set with "dvds" and "renting". Consequently, the names of keys and attributes are also restricted. We try to match the labels used in the spreadsheet with these lists to give the best possible name to each part of the model. A similar approach is used in (Ram 1995).

Intersection In the example shown before, if we try to intersect both names we get *client*. In this case this would be used as the name of the schema. If the intersection of attributes is a string in the list of words we have, it becomes the name of the table we are considering. For the cases where no name can be used, we concatenate the first letter of each attribute and use it as the name of the table/relation.

Connection table Usually, there is a table connecting other tables in a schema. The key of this table is usually composed by the keys of the tables it connects, and possibly by other attributes. This table is usually termed with the name of the relation, for example, “rent”, “sell”, “invoice row”, etc. Since this is very specific for each problem, and very difficult to infer, we name this table the concatenation of the names of the tables from which the key is composed.

Using these heuristics, the names inferred for our running are the following:

Ok	$(\{\{ok\}\},\{\})$
Quantity	$(\{\{qty\}\},\{\})$
Profit	$(\{\{move,price\}\},\{profit\})$
StoreWeek	$(\{\{store,week\}\},\{onsale\})$
Detergent	$(\{\{upc\}\},\{com\_code,description,size,case,nitem\})$
Sale	$(\{\{ok,qty,move,week,upc,store\}\},\{\})$

For this particular example, it was difficult to infer meaningful names and thus we gave names manually for all schemas, except for *Ok*. The inferred name for *Quantity* was *Qty*, for *Profit* was *MPP*, for *StoreWeek* was *SWO*, for *Detergent* was *UCDSCN* and for *Sale* was *OkQuantityProfitStoreWeekUCDSCN*.

3.3.2 The Candidate Keys

The second step of the algorithm is to compute a table with numbered candidate keys based on the schemas and candidate keys inferred from the spreadsheet’s data. Table 3.1 presents the numbered candidates keys for our running example.

To construct the table for the candidate keys, an entry is added for each attribute of a candidate key. Each row has the name of the attribute, the corresponding schema and a number. Within a schema, this number must be the same for each attribute of a candidate key but unique for each candidate key.

Schema	Attribute	Candidate Key #
Ok	ok	1
Quantity	qty	1
Profit	move	1
Profit	price	1
StoreWeek	store	1
StoreWeek	week	1
Detergent	upc	1
Sale	ok	1
Sale	qty	1
Sale	move	1
Sale	week	1
Sale	upc	1
Sale	store	1

Table 3.1: Table representing the candidate keys for the detergents example.

To better understand this table let us look at a couple of examples. As we can see in Table 3.1, the *Profit* schema has one candidate key with two attributes and so it has the same number for each attribute, 1. If we had a schema with more than one candidate key, say $(\{\{A,B\},\{C,D,E\},\{F\}\},\{G\})$, A and B would have the number 1, C, D and E would have 2, and F would have 3.

3.3.3 The Foreign Keys

The next step of our algorithm is to compute a table with foreign keys. Each entry in this table must have the schema and the attribute to which the candidate key points to, the schema and the attribute of the foreign key, and a link number. The link number represents the number of correspondences between a candidate key and foreign keys in different schemas. For example, if a candidate key has two attributes, each one corresponding to candidate keys in different schemas, the link number will be 1 for the first attribute and 2 for the second one.

The foreign keys for our running example are represented in Table 3.2.

As the reader may have noticed, there are several foreign keys in Table 3.2. In fact, there is a foreign key from each schema to *Sale*. Algorithm 5 produces this table:

Candidate Key Attribute		Foreign Key Attribute		Link #
Schema	Attribute	Schema	Attribute	
Ok	ok	Sale	ok	1
Quantity	qty	Sale	qty	1
Profit	move	Sale	move	1
Profit	price	Sale	price	1
StoreWeek	store	Sale	store	1
StoreWeek	week	Sale	week	1
Detergent	upc	Sale	upc	1

Table 3.2: Table representing the foreign keys for the detergents example.

it receives a set of schemas with candidate keys and creates a table with the existing foreign keys.

Algorithm 5 Algorithm to create the foreign keys' table.

```
input: a set of schemas with candidate keys schemas
output: table with all foreign keys table
  table = new empty table
  consider CKs from schemas in ascending order by their number of attributes
  for all ck in CKs do
    n = number of attributes in ck
    consider schemas in ascending order by their number of attributes
    for all schema in schemas do
      consider $s \in \mathscr{P}$ (schema) containing the same number of attributes as ck
      link# = 1
      for all s do
        for i=1 to n do
          if $ck\_schema\_name \neq schema\_name \wedge ck(i) \neq s(i)$ then
            /* ck_schema_name: name of the schema of the CK in use
            schema_name: name of the schema under consideration */
            add (ck_schema_name, ck(i), schema_name, s(i), link#) to table
          end if
        end for
        link# = link# + 1
      end for
    end for
  end for
  return table
```

This algorithm iterates over the set of candidate keys of the schemas in an ascending order of their number of attributes. For each candidate key, it considers the schemas in an ascending order of their number of attributes. For each of these schemas, it iterates over the elements of its power set with the same number of attributes of the candidate key under consideration. For different schemas and attributes, it adds the tuple (*ck_relation_name*, *ck_attribute*, *fk_schema_name*, fk_attribute, *link#*). The link number is incremented for each part of a schema considered.

3.3.4 The Primary Keys

Having computed the candidate keys and the foreign keys, a table with the primary key for each schema can now be computed. If the schema has only one candidate key, then it is chosen to be the primary key. If it has more then one candidate key, the selected one is the first that appears in the first column of the foreign keys' table.

Table 3.3 represents the primary keys for our running example. Notice that, Table 3.3 is a subset of the first two columns of Table 3.1.

Schema	Attribute
Ok	{ok}
Quantity	{qty}
Profit	{move, price}
StoreWeek	{store, week}
Detergent	{upc}
Sale	{ok, qty, move, week, upc, store}

Table 3.3: Table representing the primary keys for the detergents example.

For each schema, the candidate key is unique and so it was chosen to be the primary key.

3.3.5 The Relational Intermediate Directed Graph

The next step in the reverse engineering process is to produce a *Relational Intermediate Directed (RID) Graph* (Alhajj 2003). This graph includes all the relationships between a given set of schemas. Nodes in the RID graph represent schemas and directed edges represent foreign keys between those schemas. For each schema, a node in the graph is created, and for each foreign key, an edge is added to the graph.

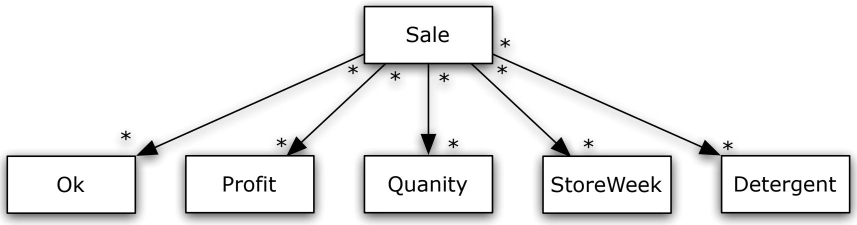


Figure 3.5: RID graph for the detergents example.

Figure 3.5 represents the RID graph for the detergents sales system. This graph can in general be improved in several ways. We explain this in the next section.

3.3.6 Optimizing the Relational Intermediate Direct Graph

It is possible to optimize and improve the RID graph by detecting relationships, that is, schemas that represent relationships connecting other schemas. In such cases, the schema is transformed into a relationship. Algorithm 6 detects these cases: it receives the foreign keys' table, the RID graph and classifies schemas as relationships, if any exist.

Algorithm 6 Algorithm to identify relationships.

```
input: the foreign keys' table $fks$ and the RID graph $rid$
output: improved RID graph
  for all relation $r$ that appears only in the third column in $fks$ do
    let $w$ be the number of links connected to $r$
    if $w \geqslant 2$ then
      let $r_1, r_2, ..., r_w$ be the relations connected to $r$
      if the only candidate of $r$ is a combination of the PKs of $r_1, r_2, ..., r_w$ then
        $r$ is a relationship between $r_1, r_2, ..., r_w$
        the cardinality is $*$ to all references of $r$
      end if
    end if
  end for
```

Since the only candidate key of *Sale* is the combination of all other schemas' primary keys, it represents a relationship between all the other schemas and is therefore transformed into a relationship. The improved RID graph is illustrated in Figure 3.6.

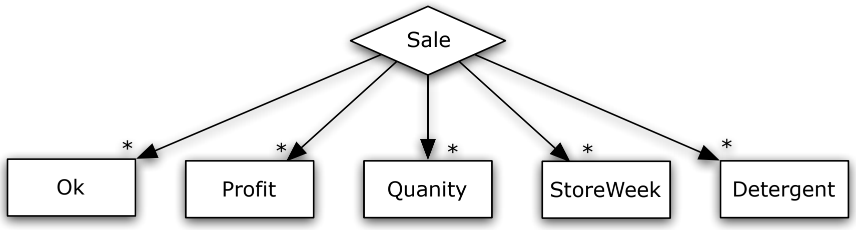


Figure 3.6: Optimized RID graph for our running example.

The RID graph shown in Figure 3.6 represents 5 entities: *Ok*, *Quantity*, *StoreWeek*, *Profit* and *Detergent*. All of them have the cardinality “*” as assigned by Algorithm 6. The relationship *Sale* connects them all.

3.3.7 The Relational Schema

Using the results from the previous sections it is now possible to compute a relational schema. We express the generated schema using standard *Entity-Relationship* (ER) diagrams (Chen 1976). We choose to express the complete database schema as an ER model since it is still considered the dominant method of conceptual data modeling (Muller 1999; Davies *et al.* 2006).

We use the improved RID graph as the basis for the ER model: each schema node is transformed in an ER entity and the each relationship node in an ER relationship. The directed arrows with “*” cardinality are transformed in *one-to-n* connections. We add the corresponding keys and remaining attributes to each relation.

Figure 3.7 illustrates the ER model for our running example.

As we can see in the ER diagram illustrated in Figure 3.7, attributes are represented by circles; the ones underlined represent the key of the corresponding entity.

Using the textual representation we described in Section 2.3, this model is represented as shown next:

Ok	$(\underline{ok})$
$Quantity$	$(\underline{qty})$
$Profit$	$(\underline{move, price}, profit)$
$StoreWeek$	$(\underline{store, week}, onsale)$
$Detergent$	$(\underline{upc}, com\_code, description, size, case, nitem)$
$< Sale >$	$(\underline{\#upc, \#move, \#price, \#store, \#week, \#qty, \#ok})$

Although ER diagrams are visually very expressive, in this thesis we prefer to use

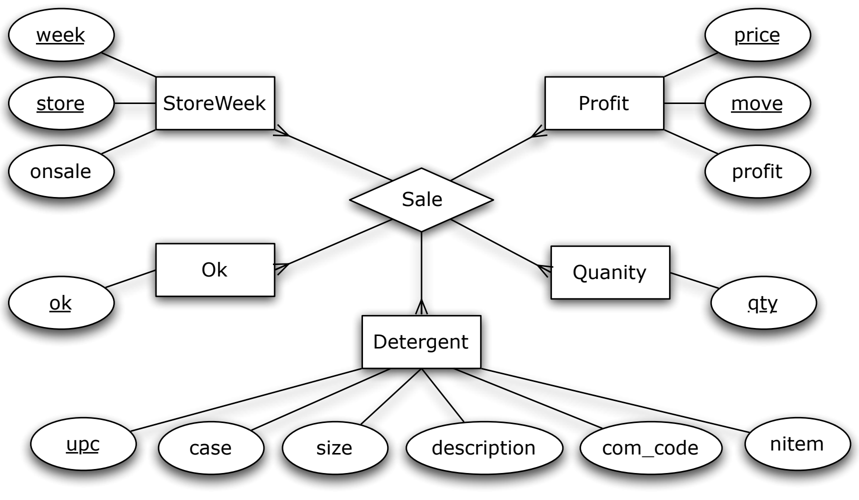


Figure 3.7: ER model specifying our running example.

the textual representation, since it is more compact, and in our case, has the same expressiveness.

3.4 Deriving a *ClassSheet* Specification

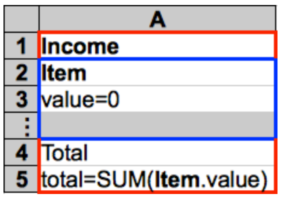
In the previous section we have described how to automatically infer a relational schema from spreadsheet data. This kind of specification is mostly used is the database realm since it is very appropriate to model data. Unfortunately, is some cases, this may not be enough for spreadsheets since this model discards, for example, the layout of a spreadsheet.

In this section, we explain in detail the steps to automatically extract a *ClassSheet* model from a spreadsheet. These specifications can model all the aspects of a spreadsheet, including its layout. We start by presenting in more detail the *ClassSheet* model.

3.4.1 *ClassSheets*

The *ClassSheet* modeling framework offers high-level, object-oriented formal models to specify spreadsheets (Engels and Erwig 2005). *ClassSheets* allow users to express business object structures within a spreadsheet using concepts from UML. From this specification, a spreadsheet application consistent with the model can be automatically generated. This new spreadsheet has mechanisms to guide the user interaction with the spreadsheet preventing in this way a significant amount of errors.

Let us look at the example illustrated in Figure 3.8, the so-called *income* sheet, consisting of a list of values, which are summed up and the result shown in a separate cell.



	A
1	**Income**
2	**Item**
3	value=0
⋮	
4	Total
5	total=SUM(**Item**.value)

Figure 3.8: An example of a *ClassSheet*: the *income* sheet.

From an object-oriented point of view, one can see a summation object, which aggregates a list of objects containing single values. Looking at the layout structure, and starting with the blue part, we have a class labeled **Item**, consisting of a value which has as default 0. In fact, this is not a single value, but a list, since the next row (between row 3 and row 4) is labeled by the ellipses.

The summation object, in red, is defined by the label **Income**, a footer with the label **Total** and an aggregation formula assigned to an attribute named **total**. Such an object-oriented extended template is called a *ClassSheet* since it defines classes together with their attributes and aggregational relationships.

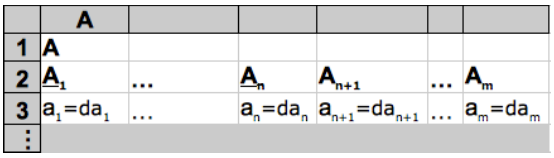
As we can see, *ClassSheets* consist of a list of attribute definitions grouped by classes and arranged on a two dimensional grid. Additional labels are used to annotate the concrete representation. References to other entries are defined by using attribute names, as shown in the SUM formula in the example.

The formal language of *ClassSheets* is defined as follows (Engels and Erwig 2005):

$$
\begin{array}{llll}
f \in Fml & ::= & \varphi \mid n.a \mid \varphi(f,\ldots,f) & (formulas) \\
b \in Block & ::= & \varphi \mid a = f \mid b \shortmid b \mid b\hat{\ }b & (blocks) \\
l \in Lab & ::= & h \mid v \mid .n & (classlabels) \\
h \in Hor & ::= & \underline{n} \mid \underline{\shortmid n} & (horizontal) \\
v \in Ver & ::= & \shortmid n \mid \underline{\shortmid n} & (vertical) \\
c \in Class & ::= & l : b \mid l : b^{\downarrow} \mid c\hat{\ }c & (classes) \\
s \in Sheet & ::= & c \mid c^{\rightarrow} \mid s \shortmid s & (sheets)
\end{array}
$$

3.4.2 Generating *ClassSheets*

The relational schema generated in Section 3.3.7 can be translated into a *ClassSheet* diagram. By default, each relation is translated into a class with the same name as the relation and a vertically expanding block. In general, for a relation A with attributes $\underline{A_1}, \ldots, \underline{A_n}, A_{n+1}, ..., A_m$ and default values $da_1, \ldots, da_n, d_{n+1}, ..., d_m$, a *ClassSheet* class/table is generated as shown in Figure 3.9[1].



	A					
1	**A**					
2	$\underline{\mathbf{A_1}}$	...	$\underline{\mathbf{A_n}}$	$\mathbf{A_{n+1}}$	...	$\mathbf{A_m}$
3	a_1=da_1	...	a_n=da_n	a_{n+1}=da_{n+1}	...	a_m=da_m
⋮						

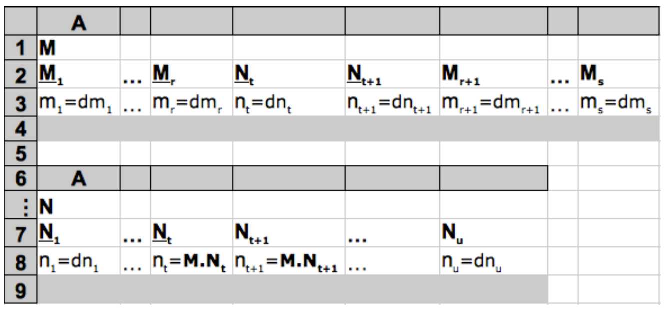
Figure 3.9: Generated class for a relation A.

This *ClassSheet* represents a spreadsheet "table" with name **A**. For each attribute, a column is created and is labeled with the attribute's name. The default values depend on the attribute's domain. This table expands vertically, as indicated by the ellipses. The key attributes become underlined labels.

A special case occurs when there is a foreign key from one relation to another. The two relations are created basically as described above but the attributes that compose the foreign key do not have default values, but references to the corresponding attributes in the other class. Let us use as an example the following relations:

$M\ (\underline{M_1, ..., M_r, N_t, N_{t+1}}, M_{r+1}, ..., M_s)$
$N\ (\underline{N_1, ..., \#N_t}, \#N_{t+1}, ..., N_u)$

The corresponding *ClassSheet* is illustrated in Figure 3.10:



	A							
1	**M**							
2	$\underline{\mathbf{M_1}}$	...	$\underline{\mathbf{M_r}}$	$\underline{\mathbf{N_t}}$	$\underline{\mathbf{N_{t+1}}}$	$\mathbf{M_{r+1}}$	...	$\mathbf{M_s}$
3	m_1=dm_1	...	m_r=dm_r	n_t=dn_t	n_{t+1}=dn_{t+1}	m_{r+1}=dm_{r+1}	...	m_s=dm_s
4								
5								
6	A							
⋮	**N**							
7	$\underline{\mathbf{N_1}}$	...	$\underline{\mathbf{N_t}}$	$\mathbf{N_{t+1}}$	...	$\mathbf{N_u}$		
8	n_1=dn_1	...	n_t=**M.N**$_t$	n_{t+1}=**M.N**$_{t+1}$	...	n_u=dn_u		
9								

Figure 3.10: Generated *ClassSheet* for relations with foreign keys.

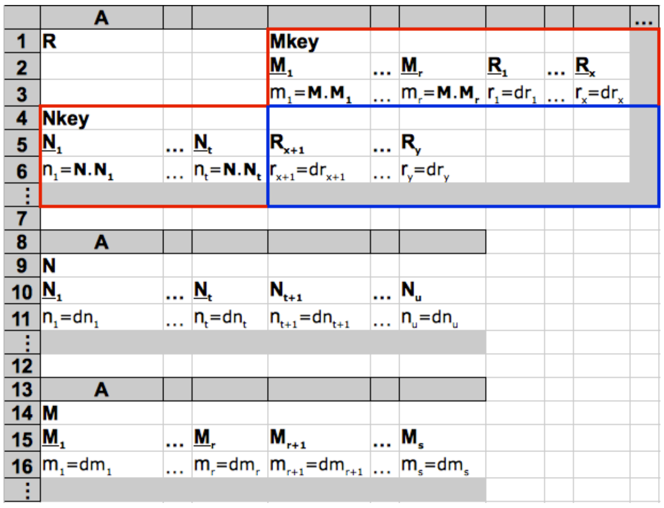
[1]We omit here the column labels, whose names depend on the number of columns in the generated table.

Relationships are treated differently and will be translated into cell classes. We distinguish between two cases: *(A)* relationships between two schemas, and *(B)* relationships between more than two schemas.

For case *(A)*, let us consider the following set of schemas:

$$\begin{array}{ll} M & (\underline{M_1, \ldots, M_r}, M_{r+1}, \ldots, M_s) \\ N & (\underline{N_1, \ldots, N_t}, N_{t+1}, \ldots, N_u) \\ <R> & (\underline{M_1, \ldots, M_r, N_1, \ldots, N_t, R_1, \ldots, R_x}, R_{x+1}, \ldots, R_y) \end{array}$$

The *ClassSheet* that is produced by this translation is shown in Figure 3.11 and explained next. For both nodes M and N a class is created as explained before (lower part of the *ClassSheet*). The top part of the *ClassSheet* is divided in two classes and one cell class. The first class, **NKey**, is created using the key attributes from the **N** class. All its values are references to **N**. For example, $n1 = \mathbf{N.N1}$ references the values in column `A` in class **N**. This makes the spreadsheet easier to maintain while avoiding insertion, modification and deletion anomalies. Class **Mkey** is created using the key attributes of the class **M** and the rest of the key attributes of the relationship R. The cell class (with blue border) is created using the rest of the attributes of the relationship R.



	A									...
1	R			Mkey						
2				M_1	...	M_r	R_1	...	R_x	
3				m_1=**M.M**$_1$	...	m_r=**M.M**$_r$	r_1=dr_1	...	r_x=dr_x	
4	Nkey									
5	N_1	...	N_t	R_{x+1}	...	R_y				
6	n_1=**N.N**$_1$	...	n_t=**N.N**$_t$	r_{x+1}=dr_{x+1}	...	r_y=dr_y				
⋮										
7										
8	A									
9	N									
10	N_1	...	N_t	N_{t+1}	...	N_u				
11	n_1=dn_1	...	n_t=dn_t	n_{t+1}=dn_{t+1}	...	n_u=dn_u				
⋮										
12										
13	A									
14	M									
15	M_1	...	M_r	M_{r+1}	...	M_s				
16	m_1=dm_1	...	m_r=dm_r	m_{r+1}=dm_{r+1}	...	m_s=dm_s				
⋮										

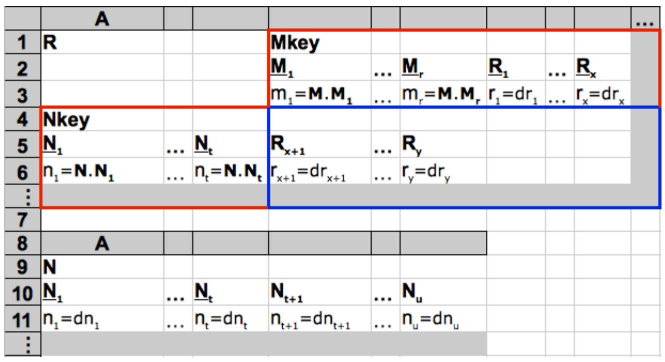
Figure 3.11: *ClassSheet* of a relationship connecting two relations.

In principle, the positions of **M** and **N** are interchangeable and we have to choose which one expands vertically and which one expands horizontally. We choose whichever combination minimizes the number of empty cells created by the cell class, that is, the number of key attributes from **M** and **R** should be similar to the number of non-key attributes of **R**. Three special cases can occur with this configuration.

The first case occurs when one of the relations **M** or **N** might have only key attributes. Let us assume that **M** is in this situation:

$$
\begin{array}{ll}
M & (\underline{M_1,\ldots,M_r}) \\
N & (\underline{N_1,\ldots,N_t},N_{t+1},\ldots,N_u) \\
<R> & (\underline{M_1,\ldots,M_r,N_1,\ldots,N_t,R_1,\ldots,R_x},R_{x+1},\ldots,R_y)
\end{array}
$$

In this case, and since all the attributes of that class are already included in the class **MKey** or **NKey**, no separated class is created for it. The resultant *ClassSheet* would be similar to the one presented in Figure 3.11, but a separated class would not be created for **M** or for **N** or for both. Figure 3.12 illustrates this situation.



	A									...
1	R			Mkey						
2				$\underline{M_1}$	...	$\underline{M_r}$	$\underline{R_1}$	...	$\underline{R_x}$	
3				m_1=**M.M**$_1$	...	m_r=**M.M**$_r$	$r_1=dr_1$	...	$r_x=dr_x$	
4	Nkey									
5	$\underline{N_1}$	...	$\underline{N_t}$	R_{x+1}	...	R_y				
6	n_1=**N.N**$_1$	...	n_t=**N.N**$_t$	$r_{x+1}=dr_{x+1}$	...	$r_y=dr_y$				
⋮										
7										
8	A									
9	N									
10	$\underline{N_1}$	...	$\underline{N_t}$	N_{t+1}	...	N_u				
11	$n_1=dn_1$	...	$n_t=dn_t$	$n_{t+1}=dn_{t+1}$	...	$n_u=dn_u$				
⋮										

Figure 3.12: *ClassSheet* where on entity has only key attributes.

The second case occurs when the key of the relationship R is only composed by the keys of M and N (defined as before), that is, R is defined as follows:

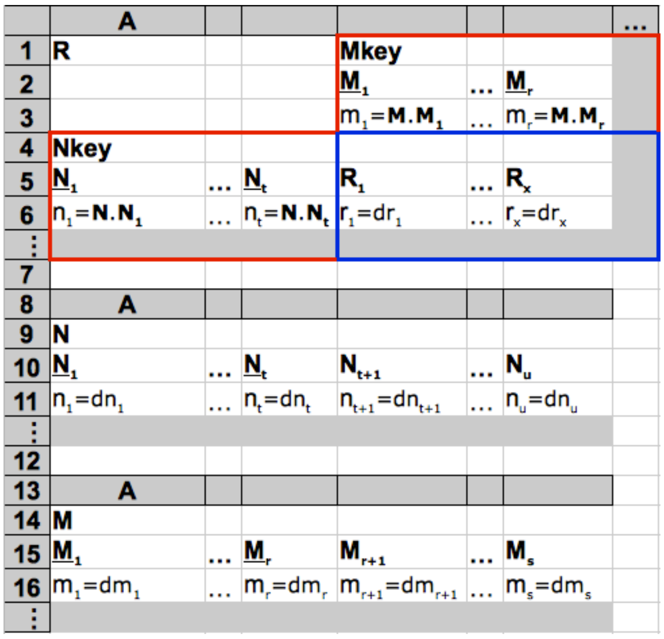
$$
\begin{array}{ll}
M & (\underline{M_1,\ldots,M_r},M_{r+1},\ldots,M_s) \\
N & (\underline{N_1,\ldots,N_t},N_{t+1},\ldots,N_u) \\
<R> & (\underline{M_1,\ldots,M_r,N_1,\ldots,N_t},R_1,\ldots,R_x)
\end{array}
$$

The resultant *ClassSheet* is shown in Figure 3.13.

The difference between this *ClassSheet* model and the general one is that the **MKey** class on the top does not contain any attribute from R: all its attributes are contained in the cell class.

Finally, the third case occurs when the relationship is composed only by key attributes as illustrated next:

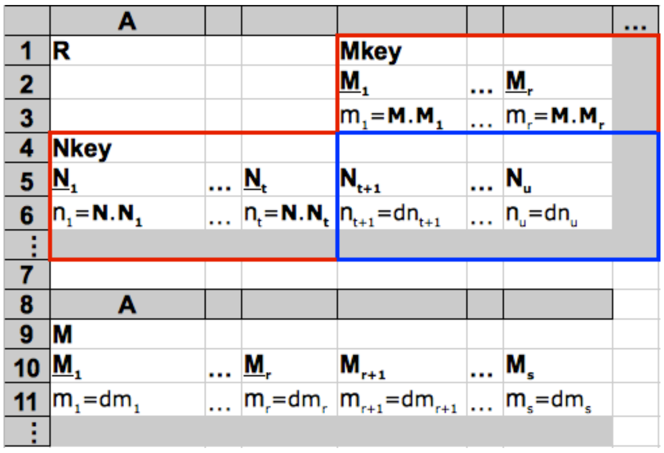
$$
\begin{array}{ll}
M & (\underline{M_1,\ldots,M_r},M_{r+1},\ldots,M_s) \\
N & (\underline{N_1,\ldots,N_t},N_{t+1},\ldots,N_u) \\
<R> & (\underline{M_1,\ldots,M_r,N_1,\ldots,N_t})
\end{array}
$$



	A						...
1	R			Mkey			
2				M_1	...	M_r	
3				m_1=M.M_1	...	m_r=M.M_r	
4	Nkey						
5	N_1	...	N_t	R_1	...	R_x	
6	n_1=N.N_1	...	n_t=N.N_t	r_1=dr_1	...	r_x=dr_x	
⋮							
7							
8	A						
9	N						
10	N_1	...	N_t	N_{t+1}	...	N_u	
11	n_1=dn_1	...	n_t=dn_t	n_{t+1}=dn_{t+1}	...	n_u=dn_u	
⋮							
12							
13	A						
14	M						
15	M_1	...	M_r	M_{r+1}	...	M_s	
16	m_1=dm_1	...	m_r=dm_r	m_{r+1}=dm_{r+1}	...	m_s=dm_s	
⋮							

Figure 3.13: *ClassSheet* of a relationship with all the key attributes being foreign keys.

In this situation, the attributes that appear in the cell class are the non-key attributes of **N** and no class is created for **N**. Figure 3.14 illustrates this case.

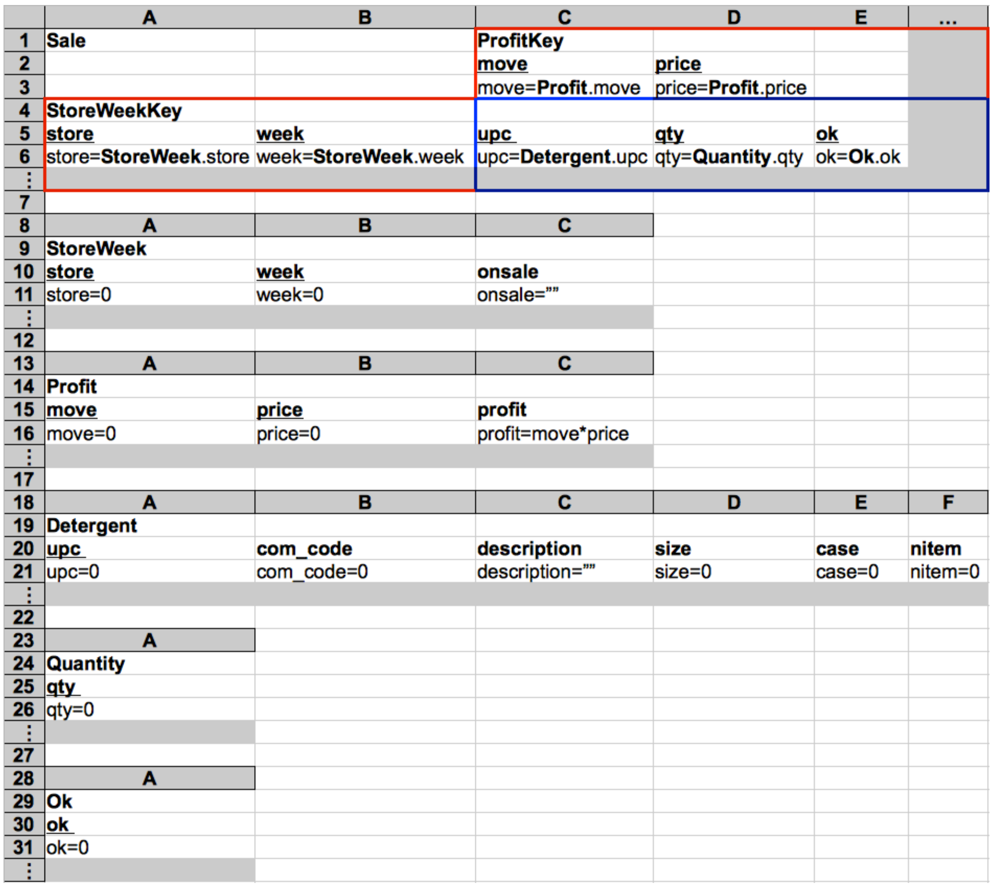


	A						...
1	R			Mkey			
2				M_1	...	M_r	
3				m_1=M.M_1	...	m_r=M.M_r	
4	Nkey						
5	N_1	...	N_t	N_{t+1}	...	N_u	
6	n_1=N.N_1	...	n_t=N.N_t	n_{t+1}=dn_{t+1}	...	n_u=dn_u	
⋮							
7							
8	A						
9	M						
10	M_1	...	M_r	M_{r+1}	...	M_s	
11	m_1=dm_1	...	m_r=dm_r	m_{r+1}=dm_{r+1}	...	m_s=dm_s	
⋮							

Figure 3.14: *ClassSheet* of a relationship composed only key attributes.

For case *(B)*, that is, for relationships between more than two tables, we choose between the candidates to span the cell class using the following criteria: (1) **M** and **N** should have small keys; (2) the number of empty cells created by the cell class should be minimal. After having chosen the two relations (and the relationship), the generation proceeds as described above. The remaining relations are created as explained in the beginning of this section.

In Figure 3.15 we present the *ClassSheet* model that is generated by our approach for the detergent application.



	A	B	C	D	E	...
1	Sale		ProfitKey			
2			move	price		
3			move=**Profit**.move	price=**Profit**.price		
4	StoreWeekKey					
5	store	week	upc	qty	ok	
6	store=**StoreWeek**.store	week=**StoreWeek**.week	upc=**Detergent**.upc	qty=**Quantity**.qty	ok=**Ok**.ok	
⋮						
7						
8	A	B	C			
9	StoreWeek					
10	store	week	onsale			
11	store=0	week=0	onsale=""			
⋮						
12						
13	A	B	C			
14	Profit					
15	move	price	profit			
16	move=0	price=0	profit=move*price			
⋮						
17						
18	A	B	C	D	E	F
19	Detergent					
20	upc	com_code	description	size	case	nitem
21	upc=0	com_code=0	description=""	size=0	case=0	nitem=0
⋮						
22						
23	A					
24	Quantity					
25	qty					
26	qty=0					
⋮						
27						
28	A					
29	Ok					
30	ok					
31	ok=0					
⋮						

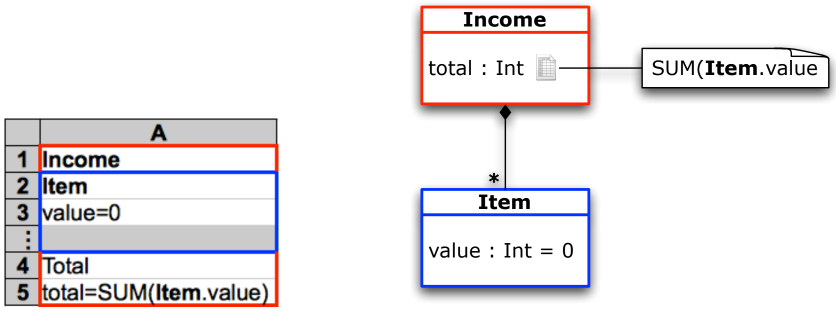
Figure 3.15: The *ClassSheet* generated by our algorithm to the detergents example.

In this case, we have a relationship, namely *Sale*, connecting several relations. From these relations, we choose *StoreWeek* and *Profit* to complete the cell class.

3.5 Deriving a UML Class Diagram

The *unified modeling language* (UML) is one of the most used languages to specify and document (software) systems (Rumbaugh *et al.* 2004). In particular, the class diagram component is very useful to design business applications. In this section we explain how to map a *ClassSheet* into one of these diagrams. We provide a class diagram as a tool to allow transformations to other paradigms, such as, for example, the objected-oriented one (Meyer 1997).

Let us first look at the *Income* example illustrated in Figure 3.16.



(a) Income *ClassSheet*. (b) Income UML class diagram.

Figure 3.16: Two models for the *income* sheet: a *ClassSheet* and a UML class diagram.

This example illustrates the similarities between a *ClassSheet* and a UML class diagram. The class **Item** is represented in UML as a class with the same name and one attribute, `value`, of type *Int* and initial value 0. The **Income** class is defined in a similar way but its attribute, `total`, is defined as the sum of the values of the **Item** class. These two classes are connected: **Income** is *composed* by **Item** that can be repeated many times.

In the following sections we systematize the transformation of *ClassSheet* models into UML class diagrams. We translate each of the elements that compose the *ClassSheet* language, as shown in Section 3.4.

3.5.1 Mapping Blocks

Remember that, blocks are defined as follows: $b \in Block ::= \varphi \mid a = f \mid b \shortmid b \mid b \hat{} b$.

φ Used in the *ClassSheet* to label cells. Since in UML we will not have this spreadsheet specificity, this is not translatable to UML.

$a = f$ Used to define attributes: the attribute a is defined by the formula f. This is mapped into a class attribute associated to the corresponding class. The attribute name is a and its associated formula is f. Since f can assume several forms, we detail each of them next. f is defined as follows: $f \in Fml ::= \varphi \mid n.a \mid \varphi(f,\ldots,f)$.

φ Used to represent default values of an attribute. This is mapped without any translation. For the class under consideration, the attribute is created as represented in Figure 3.17.

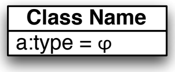


Figure 3.17: Mapping default values to UML.

$n.a$ Used to qualified access to attributes: get attribute a from class n. In this case the definition of the attribute is different since it is not defined by a default value, but through a reference to another class. Figure 3.18 represents the attribute definition for the corresponding class.

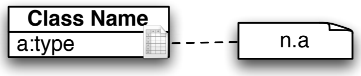


Figure 3.18: Mapping qualified access to UML.

This *ClassSheet* element is also mapped to an association between the class where it is being used and the class it references.

$\varphi(f,\ldots,f)$ Used to represent n-ary functions. The mapping is done in the same way has to qualified attribute access. The formula associated to the attribute does not change from the original one.

$b \mid b$ **and** $b\hat{}b$ All these *bs* are recursively used to compute all the attributes and labels for the classes under consideration.

3.5.2 Mapping Labels

From a class label we simply use the label itself. For $.n$ we use n; for the other cases, that is, h or v we use the corresponding horizontal or vertical label.

3.5.3 Mapping Classes

ClassSheet classes are defined as follows: $c \in Class ::= l : b \mid l : b^{\downarrow} \mid c\hat{}c$.

$l : b$ Used to create a *ClassSheet* class with label l and block b. From the label l we extract the name to the UML class as explained in Section 3.5.2. The block b is used to attach all the attributes to the just created UML class. If a class with the same name already exists, no new class is created. Instead, the attributes are attached to the existing class.

$l : b^{\downarrow}$ Used to express blocks that expand vertically. This is mapped in the same way as the previous one. The difference is in the associations: they have type $*$.

$c\,\hat{}\,c$ Both classes are handled recursively to create UML classes.

3.5.4 Mapping Sheets

Sheets are defined as follows: $s \in \mathit{Sheet} ::= c \mid c^{\rightarrow} \mid s \shortmid s$.

c This is a class, and this, it is handled as explained before.

$c^{\rightarrow}$ This class is mapped in a similar way to a regular class. The relationships it has have cardinality $*$.

$s \shortmid s$ Both sheets are recursively handled.

For our running example, Figure 3.19 illustrates the UML class diagram generated from the *ClassSheet* computed in Section 3.4.

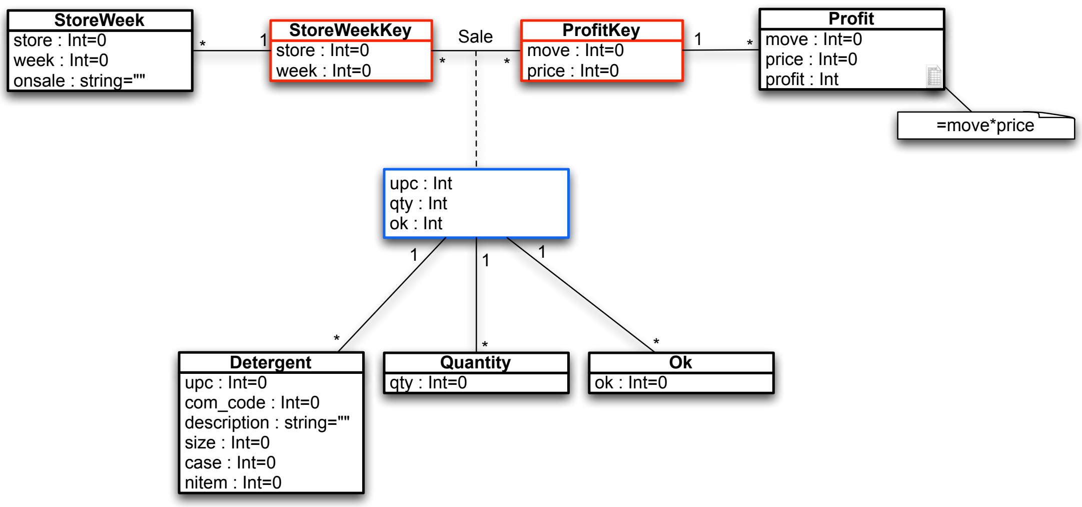


Figure 3.19: A UML class diagram specifying the detergent sale system.

3.6 Evaluation

In order to evaluate the applicability of our approach, we have implemented it in the HAEXCEL framework (see Chapter 8). Using the results of this tool, we have performed an experiment on the spreadsheets that are made available (through a CD) with (Powell and Baker 2003). This set consists of 27 spreadsheets, each containing between 1 and 16 worksheets, with a total of 121 worksheets[2]. More than half of worksheets, 66 out of the 121, contain formulas.

With this experiment we want to test whether the *ClassSheet* inference approach works in practice. Specifically, we want to know how well the system is able to identify table and relationship structures and for which kinds of spreadsheets it works and when it fails. We also want to know the quality of the *ClassSheet* models generated.

More precisely, we seek to answer the following research questions:

RQ1 In how many cases is *ClassSheet* inference applicable?

RQ2 How many of the table and relationship structures that can be identified in the data can be successfully captured by *ClassSheets* inferred by our tool?

RQ3 In which cases does *ClassSheet* inference fail?

3.6.1 Test Results

To answer the first two research questions we manually inspected all the spreadsheets to see how many tables could be identified and what relationships exist.

We present in Table 3.4 the results of this evaluation. Four of the tables in spreadsheet “c9/Options.xls” contained spreadsheet errors (reported by *Excel*); these tables were excluded from the further analysis.

Through manual inspection of the spreadsheets, we were able to identify 176 tables. In the best case, *ClassSheet* inference would be able to identify all 176 tables and create a *ClassSheet* representation for them.

The results in Table 3.4 show that the tool could identify all but 13 tables. The tool failed mainly in those cases when no layout in the spreadsheets was available. Although through manual inspection we could recognize a table structure, the tool was unable to derive sufficient constraints from layout to support the heuristics for the

[2]One spreadsheet, c6/Adbudget6.xls, was unreadable, as reported by *Excel*.

Spreadsheet	sheets	tables rec. manually	fail	bad	accept	good
c4/SS Kuniang.xls	1	2				2
c5/AdBudget.xls	8	13	6			7
c5/Delta.xls	5	5			2	3
c7/Bundy.xls	10	44			2	42
c7/Forecasting.xls	7	10	6			4
c7/Analgesics.xls	3	5		1		4
c7/Applicants.xls	6	6		2	1	3
c7/Dish.xls	2	2				2
c7/Executives.xls	6	11		2	1	8
c7/Population.xls	3	4			4	
c7/Tissue.xls	1	1			1	
c8/AdBudget8.xls	8	8		2	2	4
c8/IP.xls	7	1		1		
c8/LP.xls	16	7			2	5
c8/NLP.xls	5	5				5
c9/AdBudget9.xls	7	6		1	1	4
c9/Butson.xls	2	4			4	
c9/Data.xls	2	10				10
c9/Diffusion.xls	2	4		1		3
c9/Hastings1.xls	3	4		2		2
c9/Hastings2.xls	2	1				1
c9/Netscape.xls	5	8			7	1
c9/Options.xls	8	4	1			3
c9/Plants.xls	2	10				10
c9/Portfolio.xls	3	1				1
c9/Veerman.xls	1	0				
Total	121	176	13	12	27	124

Table 3.4: Results of the *ClassSheet* inference evaluation.

successful detection of functional dependencies. Note that, the heuristics used are the same for all the spreadsheets.

Inspection of the 163 successfully produced *ClassSheet* models suggested that they should be classified into three different levels of quality: *bad*, *acceptable*, and *good*.

A *ClassSheet* is classified as *bad* if the underlying relational model is not realistic. In some cases it is not possible to infer a model that is similar to the one an expert would create. Although from a data point of view it is correct and normalized, we consider that an expert would produce a better model. We found that 12 *ClassSheets* fall under this classification.

The classification *acceptable* was given to *ClassSheets* that do not completely characterize the corresponding table or relationship; while capturing many or most of its essential aspects, it left out some important parts. We classified 27 *ClassSheets* as *acceptable*.

Finally, a *good ClassSheet* is a model that closely represents the tables and relationships under consideration. The relational model inferred is very realistic, and the produced *ClassSheet* model is well structured. From the 163 tables that *ClassSheet* inference was able to process, the tool produced 124 good *ClassSheets*.

3.6.2 Discussion

The test results are quite encouraging: with our techniques we are able to compute *ClassSheets* for more then 92% of the existing tables. Of these models, more then 76% are classified as good.

The failure to generate good models in about 1/4th of the cases was mostly due to two facts: (1) lack of layout to inform the heuristics and (2) some functional dependencies that do hold for the models did not appear in the data.

Although our process is completely automatic, we believe that the above observations point to the fact that the method could be more effective if it was helped by a human. For example, if additional functional dependencies or headers were explicitly provided by the user, the generated models could be improved.

In fact, if in the end of the process the user is not happy with the result, model evolution, as explained in Chapter 6, can be applied so the model completely satisfies the user. The model evolution will yield a migration function to transform the original spreadsheet into the new model.

3.7 Conclusions

We have developed techniques to automatically infer relational database schemas, *ClassSheets* and UML class diagrams from spreadsheets, based on functional dependencies induced by the spreadsheet data. These models can be of great help for the maintenance of spreadsheets since knowledge about the underlying model can, for example, prevent erroneous operations. In fact, this process is quite useful specially for legacy spreadsheets that could not take advantage of the techniques based on models, such as the ones we will present in the next chapters. To automatically infer models specifying the business logic of spreadsheets is important also because it is a difficult process for users (Abraham and Erwig 2006a).

The work we presented in this chapter is closely related to the one presented in (Abraham and Erwig 2006a) where a technique to infer templates from spreadsheets is presented and evaluated. The technique presented by Abraham *et al.* is based on similarity of groups of cells and depends on the discovery of repeating patterns in spreadsheets. This technique works very well for spreadsheets that do exhibit such repeating blocks. Our approach is more suitable when relationships exist between data. Another difference is that the target modeling languages are not the same: while they only compute templates in the ViTSL language (Abraham *et al.* 2005), we can generate three different models for different purposes.

We have adapted and extended a method to infer relational models from the database realm to work with spreadsheets. The exploitation of layout information that is specific to spreadsheets was instrumental in the successful working of the technique.

An evaluation of our approach on real-world spreadsheets showed encouraging results and demonstrated that the approach is viable and should be pursued further in future work.

The developed technique is a significant contribution to spreadsheet (reverse) engineering, because it fills an important gap and allows a promising design method (*ClassSheets*) to be applied to a huge collection of legacy spreadsheets with minimal effort.

Chapter 4

Spreadsheet Edit Assistance

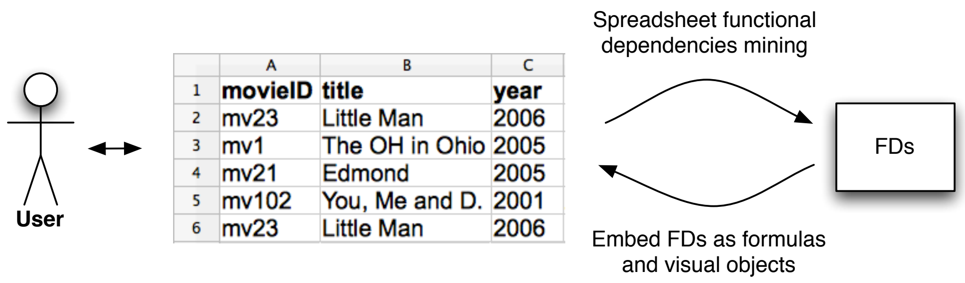
Summary

In this chapter, we demonstrate how implicit structural properties of spreadsheet data can be exploited to offer edit assistance to spreadsheet users. Our approach is based on the functional dependencies that we can infer from spreadsheet data, as explained in Chapter 2. From these functional dependencies, new formulas and visual objects are embedded in the spreadsheet to offer features for auto-completion, guarded deletion, and controlled insertion. The inference of functional dependencies and spreadsheet enhancement are carried out automatically in the background and do not disturb normal user experience.

4.1 Introduction

Recent advances in programing languages extend naive editors, to powerful language-based environments (Reps and Teitelbaum 1984; Kuiper and Saraiva 1998; van den Brand *et al.* 1999; Holzner 2004). Language-based environments use the *knowledge* of the programming language to provide the users with more powerful mechanisms to develop their programs. This knowledge is based on the *structure* and the *meaning* of the language. To be more precise, it is based on the syntactic and (static) semantic characteristics of the language. Having this knowledge about a language, the language-based environment is not only able to highlight keywords and beautify programs, but it can also detect features of the programs being edited that, for example, violate the properties of the underlying language. Furthermore, a language-based environment may also give information to the user about properties of the program under consideration. Consequently, language-based environments guide the user in writing correct programs.

Spreadsheet systems can be viewed as programming environments for end users, that is, non-professional programmers. In this chapter, we propose a technique to enhance a spreadsheet system with mechanisms to guide end users to introduce correct data. An overview of the approach is shown in Figure 4.1.


Spreadsheet functional dependencies mining
FDs
Embed FDs as formulas and visual objects
User

	A	B	C
1	**movieID**	**title**	**year**
2	mv23	Little Man	2006
3	mv1	The OH in Ohio	2005
4	mv21	Edmond	2005
5	mv102	You, Me and D.	2001
6	mv23	Little Man	2006

Figure 4.1: Edit assistance is added to an existing spreadsheet based on functional dependencies obtained by data mining.

Based on functional dependencies, a background process adds formulas and visual objects to an existing spreadsheet. To obtain these functional dependencies, we follow the approach used in language-based environments: we use the *knowledge* about the data already existing in the spreadsheet to guide end users to introduce correct data. The knowledge about the spreadsheet under consideration is based on the *meaning* of its data that we infer by using data mining and database normalization techniques as presented in Chapter 2.

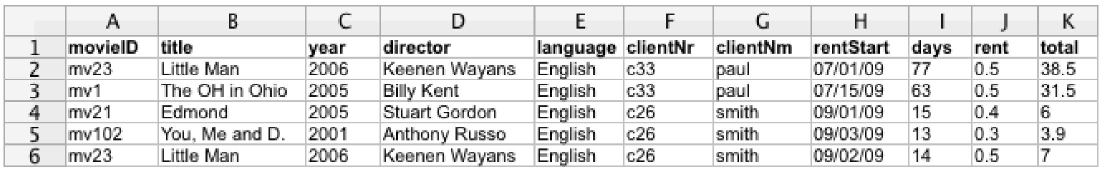
Knowing the functional dependencies that characterize the data, we construct a new spreadsheet environment that not only contains the data of the original one, but that also includes advanced features which provide information to the end user about correct data that can be introduced. We consider several types of advanced features: *bidirectional auto-completion*, *formula copying*, *non-editable columns* and *safe deletion of rows*.

Like in modern programming language environments, the refactored spreadsheet system offers the possibility of using traditional editing, that is, the introduction of data by editing each of the columns. When using traditional editing the end user is able to introduce data that may violate the functional dependencies inferred from the previous spreadsheet data. The spreadsheet environment includes a mechanism to recalculate the functional dependencies after traditional editing. These dependencies are used to guide the end user in future non-standard editing of the spreadsheet.

This chapter is organized as follows. Section 4.2 presents a motivational example used throughout the chapter. Sections 4.3 to 4.7 discuss how to embed modern programming environments in spreadsheets. In Section 4.8 we present an evaluation of our technique and Section 4.9 concludes the chapter.

4.2 Motivational Example

In order to present our approach we shall consider the following example adapted from (Berdaguer *et al.* 2007) and modeled in a spreadsheet as shown in Figure 4.2.



	A	B	C	D	E	F	G	H	I	J	K
1	**movieID**	**title**	**year**	**director**	**language**	**clientNr**	**clientNm**	**rentStart**	**days**	**rent**	**total**
2	mv23	Little Man	2006	Keenen Wayans	English	c33	paul	07/01/09	77	0.5	38.5
3	mv1	The OH in Ohio	2005	Billy Kent	English	c33	paul	07/15/09	63	0.5	31.5
4	mv21	Edmond	2005	Stuart Gordon	English	c26	smith	09/01/09	15	0.4	6
5	mv102	You, Me and D.	2001	Anthony Russo	English	c26	smith	09/03/09	13	0.3	3.9
6	mv23	Little Man	2006	Keenen Wayans	English	c26	smith	09/02/09	14	0.5	7

Figure 4.2: A spreadsheet representing a movies renting system.

The labels in the first row have the following meaning: **movieID** is a unique identifier for each movie; **title**, **year**, **director** and **language** are the common attributes of a movie; **clientNr** is a unique code for each client and **clientNm** represents the name of the client; **rentStart** denotes the starting date of the renting, **days** the number of days that the movie was rented, **rent** is the rent per day for a movie and **total** is the amount the client paid.

This spreadsheet defines a valid model to represent the information of the renting system, however, it contains redundant information. For example, the displayed data specifies the information about the movie with identifier `mv23` twice. This kind of redundancy makes the maintenance and update of the spreadsheet complex and error-prone. A mistake is easily made, for example by mistyping a name and thus corrupting the data.

As we said before, three common problems exist in redundant data: *insertion*, *modification* and *deletion anomalies*. The database community has developed techniques, such as data normalization, to eliminate such redundancy and improve data integrity.

Database normalization is based on the detection and exploitation of functional dependencies inherent in the data.

Can we leverage these database techniques for spreadsheet systems so that the system eliminates the referred anomalies by guiding the end user introducing correct data?

Based on the data in our spreadsheet example, we would like to discover the following functional dependencies, which represent the entities involved in our movie renting system: *language* of the movies, *clients* and *movies*.

$$\begin{array}{l} \mathit{language} \rightharpoonup \{\,\} \\ \mathit{clientNr} \rightharpoonup \mathit{clientNm} \\ \mathit{movieID} \rightharpoonup \mathit{title}, \mathit{year}, \mathit{director}, \mathit{rent} \end{array}$$

The reader may have noticed that, no dependencies related to formulas are included in this set. In fact, the formula columns are handled by formula copying feature, and thus, such dependencies are not necessary.

Using these functional dependencies we would like to construct a spreadsheet environment that respects those dependencies. For example, this spreadsheet would not allow the user to introduce two different movies with the same identifier *movieID*. Instead, we would like that the spreadsheet offers to the user a list of possible properties, such that he can choose the value to fill in the cell. For our running example, we would like to have a spreadsheet environment where the possible movie identifiers could be chosen from a *combo box*, that is, a button that when clicked shows a list of possible selection values, and where the value of columns *title*, *year* and *director* were automatically filled in after the selection of movie identifier is performed.

Based on functional dependencies we would like that our movie renting system spreadsheet would offer the following features to the end user:

- bidirectional auto-completion;
- formula copying;
- non-editable columns;
- safe deletion of rows;
- traditional editing;
- recalculation of the functional dependencies.

Bidirectional auto-completion We know that a functional dependency antecedent uniquely determines its consequent. Based on this principle, we wish that the spreadsheet environment automatically fills in the columns corresponding to functional dependency consequents provided the end user defines the value of the columns corresponding to the antecedents.

Moreover, if we look at a functional dependency the other way around, we can observe that, after selecting a value in the consequent, the values of the antecedent that can determine such value are restricted to a smaller set. So, we also would like that our system permits that, when selecting values from consequent columns, the values from the corresponding antecedent columns become restricted.

For example, the value of the movie number (*movieID*, column `A`) determines the values of the title (column `B`), year (column `C`), director (column `D`) and rent (column `J`). Consequently, the spreadsheet environment should be able to automatically fill in the values of columns `B`, `C`, `D` and `J`, given the value of column `A`. On the other hand, if we select the movie year `2005` we would like to see in the combo box of column `A` only the values `mv1` and `mv21`.

Formula copying A very common error done when editing spreadsheets is to replace a formula by a static value (Panko 2000). To avoid it, we make the columns with formulas non-editable. In the new rows, the formula is replaced by another formula that verifies if at least one of the cells referenced by the original formula already has a value, and when this is true, it applies the original formula.

Non-editable columns Columns that are part of the functional dependency antecedent should be non-editable. For example, column `A`, that contains the *movieID* values, is the antecedent of the movie's functional dependency. Thus, it should be protect from edition. This feature prevents the end user from introducing potential incorrect data and, thus, producing modification anomalies.

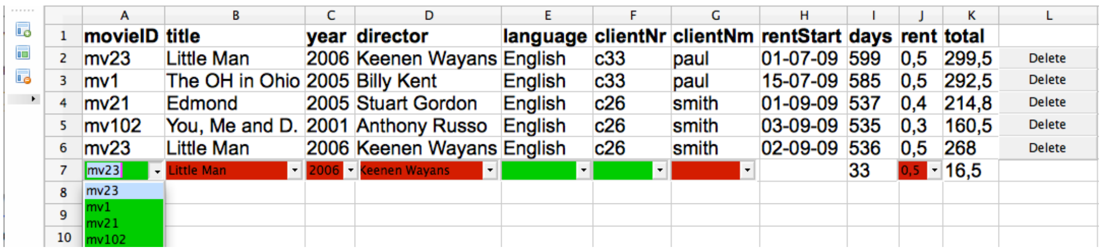
Safe deletion of rows An usual problem with non-normalized data it the deletion problem. Suppose in our running example that row 3 is deleted. All the information about the movie with identifier `mv1` (*i.e.*, *movieID*, *title*, *year* and *director*) will be lost since it is not represented elsewhere. Probably the user only wants to delete that renting transaction. To correctly delete rows in the spreadsheet, a button per row could be added as the last column of each row (column `L` in our example).

This button, when pressed, would detect if the end user was removing important information, that is, information only included in the button's row. In such a case, then, a new window warning the user should be displayed.

Traditional editing Advanced programming language environments provide both advanced editing mechanisms and traditional ones, that is, text editing. In a similar way, a spreadsheet environment should allow the user to perform traditional spreadsheet editing too. In traditional editing the end user is able to introduce data that may violate the functional dependencies that the spreadsheet data induces.

Recalculation of the functional dependencies Because standard editing allows the end user to introduce data violating the underlying functional dependencies, we would like that the spreadsheet environment would allow enable/disable the advanced features described in this section. When advanced features are disabled, the end user would be able to introduce data that violates the (previously) inferred functional dependencies. However, when the end user returns to advance editing, the spreadsheet should infer a new set of functional dependencies that would be used in future (advanced) interactions.

In this section we have described an instance of our techniques. In fact, the spreadsheet programming environment shown was automatically computed from the original spreadsheet displayed in Figure 4.2. Figure 4.3 illustrates the complete advanced edit assistant for our running example.



	A	B	C	D	E	F	G	H	I	J	K	L
1	movieID	title	year	director	language	clientNr	clientNm	rentStart	days	rent	total	
2	mv23	Little Man	2006	Keenen Wayans	English	c33	paul	01-07-09	599	0,5	299,5	Delete
3	mv1	The OH in Ohio	2005	Billy Kent	English	c33	paul	15-07-09	585	0,5	292,5	Delete
4	mv21	Edmond	2005	Stuart Gordon	English	c26	smith	01-09-09	537	0,4	214,8	Delete
5	mv102	You, Me and D.	2001	Anthony Russo	English	c26	smith	03-09-09	535	0,3	160,5	Delete
6	mv23	Little Man	2006	Keenen Wayans	English	c26	smith	02-09-09	536	0,5	268	Delete
7	mv23	Little Man	2006	Keenen Wayans					33	0,5	16,5	
8	mv23											
9	mv1											
10	mv21											
	mv102											

Figure 4.3: Complete edit assistant environment to the movies example.

On the left part of Figure 4.3 we can see three buttons representing the features just described. From the top, the first button enables edit assistance calculating the functional dependencies and creating the new environment; the second button recalculates the dependencies and the possible new objects; the last button disables the advanced editing assistance.

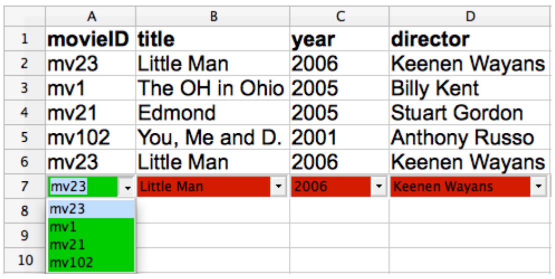
In the following sections we will present in detail the technique to perform such an automatic spreadsheet refactoring.

4.3 Bidirectional Auto-completion

This section presents techniques to refactor spreadsheets into powerful spreadsheet programming environments. The functional dependencies induced by the data included in the original spreadsheet are the building blocks for such a refactoring. In fact, the spreadsheet refactoring is implemented as the embedding of the functional dependencies in the spreadsheet. This embedding is modeled in the spreadsheet itself by standard formulas and visual objects: additional formulas and visual objects are included in the spreadsheet to guide the user to introduce correct data.

In the bidirectional auto-completion mechanism, when the user chooses one antecedent value of a functional dependency, the consequent values are automatically *filled in*. For example, the movie's code column, *movieID*, determines the movie's title, year, director and rent. Consequently, the spreadsheet environment is able to automatically fill in the values of the columns *title*, *year*, *director* and *rent* given the value of column *movieID*. This happens only for existing movies. For new movies the user must introduce all the data via standard editing.

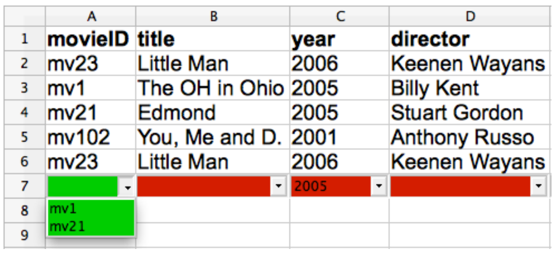
As shown in Figure 4.4, if the user selects the movie's code `mv23`, the programming environment automatically fills in the title `Little Man`, year `2006` and so on.



	A	B	C	D
1	**movieID**	**title**	**year**	**director**
2	mv23	Little Man	2006	Keenen Wayans
3	mv1	The OH in Ohio	2005	Billy Kent
4	mv21	Edmond	2005	Stuart Gordon
5	mv102	You, Me and D.	2001	Anthony Russo
6	mv23	Little Man	2006	Keenen Wayans
7	mv23	Little Man	2006	Keenen Wayans
8	mv23			
9	mv1 mv21			
10	mv102			

Figure 4.4: Selecting possible values of columns using a combo box.

By selecting a value of a consequent column the programming environment filters all other antecedent columns in the functional dependency. Figure 4.5 presents such mechanism where the end user selected the year 2005 of a movie (a consequent column) and the combo box of the movie's id only contains the movies `mv1` and `mv21`.



	A	B	C	D
1	**movieID**	**title**	**year**	**director**
2	mv23	Little Man	2006	Keenen Wayans
3	mv1	The OH in Ohio	2005	Billy Kent
4	mv21	Edmond	2005	Stuart Gordon
5	mv102	You, Me and D.	2001	Anthony Russo
6	mv23	Little Man	2006	Keenen Wayans
7	🟩	🟥	2005	🟥
8	mv1			
9	mv21			

Figure 4.5: Back propagation of the auto-completion feature.

Next, we present the details of the embedding of the functional dependencies both as spreadsheet formulas and as visual objects.

4.3.1 Generating Visual Objects

Because our embedding relies on introducing visual objects, we need to carefully choose our target spreadsheet system, since they differ in the scripting language used to express the visual objects and their interaction. The spreadsheet system chosen was *Calc* (OOoAuthors 2010a) from the *OpenOffice.org* (OOoAuthors 2010b) suite because it is a free, open source and platform independent product. We chose the BASIC (Pitonyak 2004) scripting language to implement it since it is a very simple language and can easily be migrated to other languages such as *Visual Basic for Applications* (Roman 2002), the *Excel* (Campbell 1985; O'Leary 2008) scripting language.

Let us consider the dependency *clientNr* $\rightharpoonup$ *clientNm* from our running example. In the spreadsheet, *clientNr* is in column `F` and *clientNm* in column `G`. To embed this functional dependency in the spreadsheet we first introduce a combo box containing the existing values in column `F` and another in `G`. This is achieved as follows: let *minr* be the very next row after the existing data in the spreadsheet, *maxr* the last row in the spreadsheet, and $r1$ the first row with already existing data. Each dependency $a_1 \ldots a_n \rightharpoonup c_1 \ldots c_m$, with $a_1, \ldots, a_n, c_1, \ldots, c_m$ column indexes of the spreadsheet induces a set of combo boxes defined as follows:

$$\begin{aligned}
&\forall\, c \in \{a_1, \ldots, a_n\}, r \in \{minr, maxr\}: \\
&\quad S\,(c,r) = combobox := \{\; linked\_cell := (c,r)\;; \\
&\qquad\qquad\qquad\qquad\qquad source\_cells := (c,r1) : (c,r-1)\;; \\
&\qquad\qquad\qquad\qquad\qquad backgoundColor := green\;\}
\end{aligned}$$

Notice that, this formula is for the columns composing the functional dependency antecedent. To the consequent columns the combo boxes are defined as follows:

$$
\begin{aligned}
&\forall\, c \in \{c_1,\dots,c_m\}, r \in \{minr, maxr\}:\\
&\quad S\,(c,r) = combobox := \{\ linked\_cell := (c,r)\ ;\\
&\qquad\qquad\qquad\qquad\qquad\ source\_cells := (c, r1) : (c, r-1)\ ;\\
&\qquad\qquad\qquad\qquad\qquad\ backgoundColor := red\ \}
\end{aligned}
$$

For each attribute in a functional dependency, a combo box is generated. Its *linked cell* is the cell where we want the value, that is, the cell where it is positioned. The *source cells*, that is, its possible choices, are all the cells in the same column with a lower row index. The color is used to distinguish the antecedent from the consequent columns.

To implement this we developed a front-end written in BASIC and a back-end in HASKELL. The front-end sends to the back-end the entire spreadsheet and it responds with the antecedent and consequent combo boxes and the formulas (explained later on). This result will be mainly handled by the front-end function *createCB*. It starts by creating and positioning a drawing zone to place the combo box (pink code). Then, it creates the combo box itself (blue code) and finally it creates the source cells (red code) and the linked cell (green code). The following code implements (part of) *createCB*:

```
Sub createCB (c as int, r as int, minr as int, maxr as int, x as int, y as int, bgc)
    oControlShape . setPosition (x, y)
    oCtrlMd = oDoc . createInstance ("ComboBox")
    CellRange = Sheet . getCellRangeByPosition (c, minr, c, maxr)
    oCellRangeListSource = oDoc . createInstanceWithArguments (
      "CellRangeListSource", CellRange . RangeAddress)
    oCtrlMd . setListEntrySource (oCellRangeListSource)
    oCtrlMd . BackgroundColor = bgc
    oLinkedCell . Sheet = oSheet . RangeAddress . Sheet
    oLinkedCell . Row = r
    oLinkedCell . Column = c
    oNamedValue . Value = oLinkedCell
    oCVB = oDoc . createInstance ("com.sun...CellValueBinding")
    oCtrlMd . setValueBinding (oCVB . Initialize (Array (oNamedValue)))
```

Notice that, the red, the green and the gray code correspond to the definition presented above in the same colors.

4.3.2 Generating Spreadsheet Formulas

To produce the bidirectional auto-completion feature we must generate some formulas. Let us consider once more the functional dependency *clientNr* ⇀ *clientNm* from our running example. In the spreadsheet, *clientNr* is in column `F` and *clientNm* in column `G`. To achieve auto-completion, we introduce the following formula in cell `G7`:

$$S\,(G,7) = \textbf{if}\,(\textbf{isna}\,(\textbf{vlookup}\,(F7;F2:G6;2;0));\texttt{""};\textbf{vlookup}\,(F7;F2:G6;2;0))$$

The formula **if** is the usual conditional control structure. **vlookup** searches for its first argument in the first column of its second argument (an array of columns). When it finds a match, it returns the value in the cell in the same row and in the column (given in the second argument) indexed by the third argument. The last argument states if the column to search is sorted or not. **isna** verifies if its argument is available. For example, if the **vlookup** fails, it returns a "non available" result. For more information on these and other formulas, the reader is referred to (OOoAuthors 2010a).

When a value is inserted in the client's number column (`F`), **vlookup** searches for the corresponding value in the client's name column (`G`). If something is found (checked by **isna**), it means that this correspondence exists. In this case, it returns the corresponding name (second **vlookup**). Otherwise, it returns the empty string.

We have just presented a particular case of the formula induced by a functional dependency. Next, we present the general formula:

$$\begin{array}{l}
\forall\, c \in \{c_1,\ldots,c_m\}, \forall\, r \in \{minr,\ldots,maxr\}: \\
\quad S\,(c,r) = \textbf{if}\;(\;\textbf{if}\;(\;\textbf{isna}\;(\textbf{vlookup}\;(\;(a_1,r),(a_1,r1):(c,r-1),r-a_1+1,0)), \\
\qquad\qquad\qquad \texttt{""}, \\
\qquad\qquad\qquad \textbf{vlookup}\;(\;(a_1,r),(a_1,r1):(c,r-1),r-a_1+1,0)) \\
\qquad\qquad == \\
\qquad\qquad \textbf{if}\;(\;\textbf{isna}\;(\textbf{vlookup}\;((a_2,r),(a_2,r1):(c,r-1),r-a_2+1,0)), \\
\qquad\qquad\qquad \texttt{""}, \\
\qquad\qquad\qquad \textbf{vlookup}\;((a_2,r),(a_2,r1):(c,r-1),r-a_2+1,0)) \\
\qquad\qquad == \ldots == \\
\qquad\qquad \textbf{if}\;(\;\textbf{isna}\;(\textbf{vlookup}\;((a_n,r),(a_n,r1):(c,r-1),r-a_n+1,0)), \\
\qquad\qquad\qquad \texttt{""}, \\
\qquad\qquad\qquad \textbf{vlookup}\;((a_n,r),(a_n,r1):(c,r-1),r-a_n+1,0)), \\
\qquad\qquad \textbf{vlookup}\;((a_1,r),(a_1,r1):(c,r-1),r-a_1+1,0), \\
\qquad\qquad \texttt{""})
\end{array}$$

This formula is created for each functional dependency to embed in the spreadsheet. Each *if* (in red) inside the main *if* (in green) is responsible for checking an antecedent attribute. In the case of an antecedent column value is chosen, **isna** (**vlookup** (...)), the formula calculates the corresponding consequent column value, **vlookup** (...). If the values chosen by the antecedent columns are all equal, then the found value is used in the consequent column.

An interesting case occurs when, for example, we have the functional dependencies $A \rightharpoonup B$ and $B \rightharpoonup C$. When the user inserts a value in column A, column B is automatically filled in. Since now column B also as a value, this causes the mechanism to automatically fill in column C too.

These formulas are generated in the HASKELL back-end using the functions shown next. For each consequent column it generates an inner *if* (code in red color) generating then the global *if* condition (green color).

```
genLookup :: Sheet Fml → FD String → [(String, Int, Int)]
genLookup sh (ant, cons) = let antpos = map (whereLabel sh) (atts ant)
                               conspos = map (whereLabel sh) (atts cons)
                           in map (genLU antpos) conspos
```

The auxiliary functions are defined as follows:

```
genLU :: [(Col, Row, Row)] → (Col, Row, Row) → ([Char], Col, Row)
genLU ant (col, minr, maxr) =
  let vls = map (genVLS col maxr) ant
      vls' = map (λv → "IF(ISNA(" ++ v ++ ");\"\";" ++ v ++ ")") vls
      vls'' = drop 2 (foldr (λv r → " = " ++ v ++ r) "" vls')
  in ( "=IF(" ++ vls'' ++ "; " ++ head vls ++ "; \"\")", col, maxr + 1)

genVLS :: Col → Row → (Col, Row, Row) → [Char]
genVLS col maxr (cola, row1a, row2a) =
  let  a1 = show (Indx cola (maxr + 1))
       a2 = "$" ++ intersperse '$' (show (Indx cola row1a))
       a3 = show (Indx col row2a)
  in "VLOOKUP(" ++ a1 ++ "; " ++ a2 ++ ":" ++ a3 ++ "; " ++
       show (col − cola + 1) ++ "; " ++ "0)"
```

The function *whereLabel* receives an attribute from a functional dependency and a spreadsheet and returns a triple with the column in which it is, the row after the label and the last row with data.

Notice that, the colors in this HASKELL fragment are coincident with the colors in the general formula present above.

The bidirectional use of functional dependencies can be embedded straightforwardly by using standard spreadsheet formulas. It works as follows: if a value is chosen in one of the non-key columns, the system will use this selected value to remove from the corresponding key combo boxes the values that do not determine the selected value.

4.4 Formula Copying

A very common error done when editing spreadsheets is to replace a formula by a static value (Panko 2000). To avoid this, we make the columns with formulas non-editable. In the new rows, the formula is replaced by another formula that verifies if at least one of the cells referenced by the original formula already has a value, and when this is true it applies the original formula.

If the user needs to change the formula itself, that is, its definition, the traditional mode must be activated (see Section 4.7).

As expected, this feature only works for regular formulas, that is, formulas that are virtually the same in all rows. Two formulas in two vertically consecutive cells are equal if the one in the row with greater index is equal to the other except the row indexes which are incremented by one unit. This is a very common way of using formulas in spreadsheets and is usually done dragging a formula throughout part of a column (or row). For example, the cell `K2` has the formula `=J2*I2` and the cell `K3` the formula `=J3*I3`. The same happens to all the other rows and thus this column is not editable in our running example.

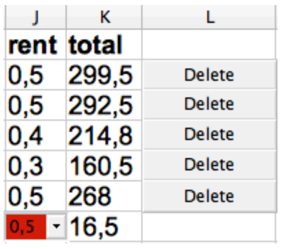
The following formula defines the first empty cell in column `K`, namely `K7`:

$$= \textbf{if}\ (\wedge\ (\textbf{isBlank}\ (I7); \textbf{isBlank}\ (J7)); \texttt{""}; I7 * J7)$$

It verifies if any of the cells used in the formula is different from blank, and, in that case, it returns the result from the original formula.

4.5 Safe Deletion

As we said before, a common problem with non-normalized data is the deletion anomaly. This problem happens because there is no separation of the entities involved in the spreadsheet. When data is normalized, the entities are isolated and thus this kind of problems do not occur. To prevent this problem it is essential to correctly delete rows in the spreadsheet. To achieve such safety, a button is added to each row, in the last column, of the spreadsheet. Figure 4.6 illustrates such buttons for our running example.



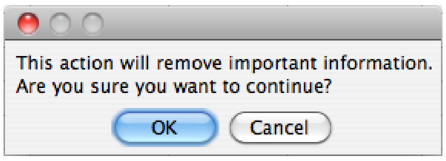
J	K	L
rent	total	
0,5	299,5	Delete
0,5	292,5	Delete
0,4	214,8	Delete
0,3	160,5	Delete
0,5	268	Delete
0,5	16,5	

Figure 4.6: The delete button is used to prevent the user for deleting crucial information.

The buttons work as follows: for each functional dependency $s, \ldots, t \rightharpoonup u, \ldots, v$ each button checks, on its corresponding row, the columns that are part of the antecedent, $s, \ldots, t$. For each antecedent column, it verifies if the value that is being removed is the last one, proceeding as follows: let $c \in \{s, \ldots, t\}$, r be the button row, $r1$ be the first row of column c with data and rn be the last row of column c with data. The test is defined using the following formula:

$$\textbf{if}\ (\textbf{isLast}\ ((c,r),(c,r1):(c,rn)), \textbf{showMessage}, \textbf{deleteRow}\ (r))$$

The function **isLast** was defined by us and verifies if the row that is being deleted contains the last entry for some of the entities in that row. If the value is the last one, the spreadsheet warns the user, **showMessage**, as can be seen in Figure 4.7.



This action will remove important information.
Are you sure you want to continue?

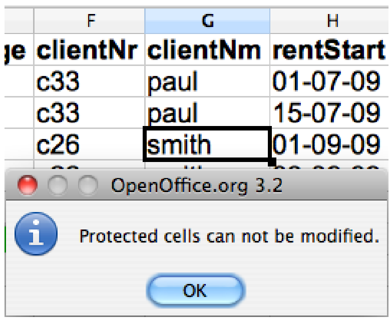
Figure 4.7: Window to warn the end user that crucial information may be deleted.

If the user presses the `OK` button, the row will be removed, **deleteRow**. In the other case, `Cancel`, no action will be performed. In the case the value is not the last one, the row will be removed, **deleteRow**.

For example, in column *movieID* of our running example, the row 3 contains the only data about the movie with code `mv1`. If the user tries to delete such row, the warning message will be triggered.

4.6 Non-editable Columns

To prevent modification anomalies, we protect some columns from being edited. Figure 4.8 illustrates such restriction.



F	G	H
clientNr	clientNm	rentStart
c33	paul	01-07-09
c33	paul	15-07-09
c26	smith	01-09-09

OpenOffice.org 3.2

Protected cells can not be modified.

Figure 4.8: Columns are not editable to prevent the modification anomalies.

Suppose the user changes in our running example cell `B2`, that is, the title of movie `Little Man`. If the user does not change the cell `B6` (the other cell with the same title) or changes it in a different way, the data will be corrupted. Moreover, the functional dependencies will not be respected anymore.

Thus, a functional dependency $s,\ldots,t \rightharpoonup u,\ldots,v$ induces that columns $u,\ldots,v$ become non-editable. That is to say that columns that are part of the consequent of a functional dependency are non-editable. When the user is inserting new rows, this restriction may not apply; if he is inserting data where the antecedent is new, the consequent cannot be automatically filled in. In these cases, the user can add new data to this kind of columns.

In case the end user needs to change the value of such protected columns, we provide traditional editing as described in the next section.

4.7 Traditional Editing

Advanced programming language environments provide both advanced editing mechanisms and traditional ones, that is, text editing. In a similar way, the generated spreadsheet environment allows the user to perform traditional spreadsheet editing too. It provides a mechanism to enable/disable the advanced features described in the previous sections. When advanced features are disabled, the end user is able to introduce data that violates the (previously) inferred functional dependencies. However, when the end user returns to advanced editing, the spreadsheet infers new functional dependencies that will be used in future interactions.

4.8 Evaluation

In order to evaluate the applicability of our approach, we have performed an experiment on the EUSES Corpus (Fisher and Rothermel 2005). It contains more than 4500 spreadsheets gathered from different sources and developed for different domains. These spreadsheets are assigned to eleven different categories. Among the spreadsheets in the corpus, only 4.4% contain macros, 2.3% contain charts, and about 56% do not have formulas being only used to store data.

In our experiment we have selected the first ten spreadsheets from each of the eleven categories of the corpus. We then applied our tool to each spreadsheet, with different results (see also Table 4.1) as explained next:

- A few spreadsheets failed to parse. This was due to glitches in the *Excel* to *OpenOffice.org* conversion.
- Some spreadsheets were parsed, but no tables could be recognized in them, that is, their users did not adhere to any of the supported layout conventions. We support the layout conventions presented in the UCheck project (Abraham and Erwig 2007b). This was the case for about 30% of the spreadsheets in our selection.
- The other spreadsheets were parsed, tables were recognized, and edit assistance was generated for them.

We will focus on the last two groups in the upcoming sections.

4.8.1 Processed Spreadsheets

The results of processing our sample of spreadsheets from the EUSES corpus are summarized in Table 4.1. The rows of the table are grouped by category as documented in the corpus. After the first column with the name of each spreadsheet, the following three columns contain size metrics of the spreadsheets. They indicate how many tables were recognized, how many columns are present in these tables, and how many cells.

File name	Recog. tables	Nr. Cols.	Cells	SSFun	Auto-compl. & safe delete	Filter
cs101						
Act4_023_capen	5	24	402	0	0	0
act3_23_bar...	6	21	84	1	1	0
act4_023_ba...	6	23	365	0	0	0
meyer_Q1	2	8	74	0	0	0
posey_Q1	5	23	72	0	0	0
database						
%5CDepart...	2	4	3463	0	0	0
00061r0P80...	23	55	491	3	4	4
00061r5P80...	30	83	600	9	9	5
0104Texas...	5	7	77	1	1	1
01BTS_fr...	52	80	305	5	5	0
03-1-rep...	20	150	1599	13	16	3
filby						
BROWN	5	14	9047	1	1	1
financial						
03PFMJOU...	15	65	242	0	0	0

Continues on the next page

Table 4.1: **– continued from previous page**

File name	Recog. tables	Nr. Cols.	Cells	SSFUN	Auto-compl. & safe delete	Filter
03Q4fins...	100	288	948	15	17	17
10-formc	12	20	53	6	6	0
111802...	166	200	356	6	6	0
forms3						
ELECLAB3...	1	4	44	0	0	0
burnett-cloc...	3	8	14	0	0	0
chen-heapS...	1	2	24	0	0	0
chen-insert...	1	2	22	0	0	0
chen-lcsTimes	1	2	22	0	0	0
chen-quickS...	1	2	24	0	0	0
cs515_npe...	7	9	93	0	0	0
cs515_pol...	6	12	105	0	0	0
cs515_run...	2	6	45	0	0	0
grades						
0304deptcal	11	41	383	17	18	9
03_04ballots1	4	20	96	2	2	2
030902	5	20	110	0	0	0
031001	5	20	110	0	0	0
031501	5	15	51	1	1	1
homework						
01_Intro_...	6	15	2115	0	0	0
01readsdis	4	16	953	4	4	3
02%20fbb...	1	7	51	0	0	0

Continues on the next page

Table 4.1: **– continued from previous page**

File name	Recog. tables	Nr. Cols.	Cells	SSFUN	Auto-compl. & safe delete	Filter
022timeli...	28	28	28	0	0	0
026timelin...	28	28	30	0	0	0
03_Stocha...	4	6	48	0	0	0
04-05_pro...	79	232	2992	0	0	0
inventory						
02MDE_fra...	50	83	207	6	6	0
02f202assi...	37	72	246	2	2	0
03-1-rep...	5	31	111	6	6	5
03singap...	9	45	153	4	4	2
0038	10	22	370	0	0	0
modeling						
%7B94402d...	1	3	561	0	0	0
%EC%86%9...	1	10	270	7	8	4
%EC%9D%9...	1	7	1442	4	4	5
%EC%A1%...	2	17	534	5	8	4
%ED%99%9...	3	7	289	4	4	1
0,10900,0-0...	4	14	6558	6	7	4
00-323r2	24	55	269	7	8	6
00000r6xP...	3	13	3528	4	5	3
003_4	25	50	2090	0	0	0

Table 4.1: Results of processing the selected spreadsheets.

The fifth column shows how many functional dependencies SSFUN generated for the recognized tables. The last two columns are metrics on the generated edit assistance. In some cases, no edit assistance was generated, indicated by zeros in these columns. This situation occurs when no (non-trivial) dependencies are induced from the recognized tables. In other cases, the two columns indicate for how many columns *bidirectional auto-completion/safe deletion* and *filtering* was generated, respectively.

For example, for the first spreadsheet of the *grades* category, bidirectional auto-completion/safe deletion has been activated for 18 columns, and filtering has been applied to 9 columns. Notice that, for the categories *jackson* and *personal*, no results were obtained due to absent or unrecognized layout conventions or to the size of the spreadsheets.

4.8.2 Observations

On the basis of these results for our sample of spreadsheets, a number of interesting observations can be made. For some categories, edit assistance is successfully added to almost all spreadsheets (for example, *inventory* and *database*), while for others almost none of the spreadsheets lead to results (for example, the *forms/3* category). The latter may be due to the small sizes of the spreadsheets in this category. The percentage of columns for which auto-completion was generated varies. The highest percentage was obtained for the second spreadsheet of the *modeling* category, with 8 out of 10 columns (80%). On the other hand, the second of the *grades* category gets edit assistance for only 2 out of 20 columns (10%). The percentage for columns with filtering, also varies. The highest percentage is obtained by the third spreadsheet in the *modeling* category with 5 out of 7 (71%). On the other hand, the last spreadsheet in the *grades* category gets 1 out of 15 columns with filtering (6.7%).

4.8.3 Discussion

Our experiment justifies two conclusions. Firstly, the tool is able to successfully add edit assistance to a series of non-trivial spreadsheets. Secondly, in the enhanced spreadsheets a large number of columns are generally affected by the generated edit assistance, which indicates that the user experience can be impacted in a significant manner. Thus, a validation experiment can be started to evaluate how users experience the additional assistance and to which extent their productivity and effectiveness can be improved. In Chapter 7 we will present an empirical study performed by real end users to validate our approach.

4.9 Conclusions

We have demonstrated how implicit structural properties of spreadsheet data can be exploited to offer edit assistance to spreadsheet users. To discover these properties, we have made use of our previously developed approach for mining, filtering and normalizing functional dependencies from spreadsheets. On this basis, we have made the following contributions:

- Derivation of formulas and visual elements that capture the knowledge encoded in a set of functional dependencies;
- Embedding of these formulas and visual elements into the original spreadsheet in the form of features for auto-completion, guarded deletion, and controlled insertion;
- Integration of the algorithms for manipulation of functional dependencies, for derivation of corresponding formulas and visual elements, and for their embedding into an extension for spreadsheet environments.

A spreadsheet environment enhanced with our extension compensates to a significant extent for the lack of the structured programming concepts in spreadsheets. In particular, it assists users to prevent common modification and deletion anomalies during edit actions.

There are several extensions of our work that we would like to explore. The algorithms running in the background need to recalculate the functional dependencies and the ensuing formulas and visual elements every time new data is inserted. For larger spreadsheets, this recalculation may incur waiting time for the user. Several optimizations of our algorithms can be attempted to eliminate such waiting times. In particular, incremental recomputation could be used.

Our approach could be integrated with complementary approaches to cover a wider range of possible user errors. In particular, the work of Abraham *et al.* (Abraham and Erwig 2006a,b) for preventing range, reference, and type errors could be combined with our work for preventing data loss and inconsistency.

Chapter 5

Migration of Spreadsheets

Summary

This chapter presents techniques to transform spreadsheets into relational databases and back. A set of data refinement rules is introduced to map a tabular data type into a relational database schema. Having expressed the transformation of the two data models as data refinements, we obtain for free the functions that migrate the data. We use well-known relational database techniques to optimize and query the data. Because data refinements define bidirectional transformations we can map such databases back to optimized spreadsheets.

5.1 Introduction

Spreadsheets are applications created by single end users, without planning ahead of time for maintainability or scalability. Still, after their initial creation, many spreadsheets turn out to be used for storing and processing increasing amounts of data and supporting increasing numbers of users over long periods of time. To turn such spreadsheets into database-backed multi-user applications with high maintainability is not a smooth transition, requiring substantial time and effort.

In this chapter, we develop techniques for smooth transitions between spreadsheets and relational databases. The basis of these techniques is the fundamental insight that spreadsheets and relational databases are formally connected by data refinement rules. To find this relationship we use functional dependencies inferred from the spreadsheet data as explained in Chapter 2. These functional dependencies can be exploited to derive a relational database schema as described in Chapter 3. We then apply data calculation laws to the derived schema in order to reconstruct a sequence of refinement steps that connects the relational database schema back to the tabular spreadsheet. Each refinement step at the schema level is witnessed by bidirectional conversion steps at the data level, allowing data to be converted from spreadsheet to database and vice versa, as shown in Figure 5.1.

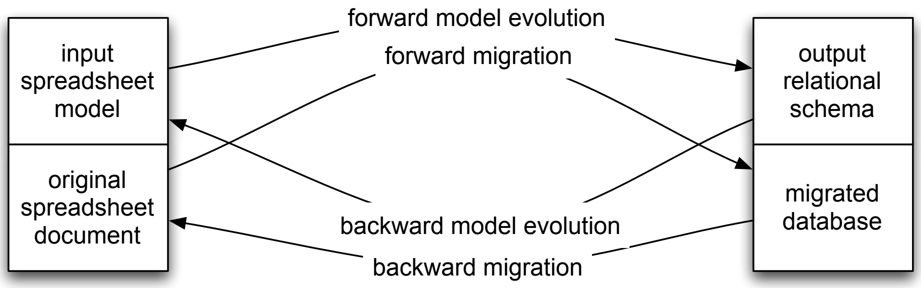


Figure 5.1: Model evolution is coupled with migration function back and forward.

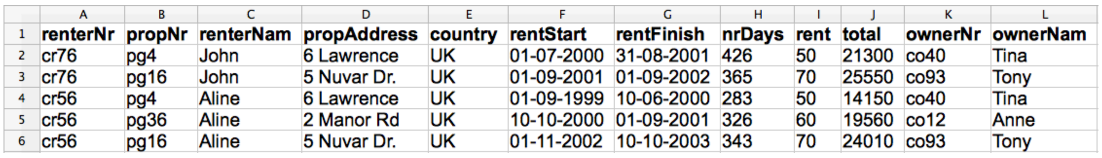
Our approach is to employ techniques for bidirectional transformation of types, values, functions, and constraints (Visser 2008), based on data refinement theory (Morgan and Gardiner 1990; Oliveira 1990, 2008).

We have implemented data refinement rules in HASKELL for converting between tabular and relational data types as a framework. This framework migrates not only values between the two data models, but also formulas.

This chapter is organized as follows. Section 5.2 presents a motivational example used throughout the chapter. In Section 5.3 we define data refinements and a framework for constraint-aware two-level transformation. Section 5.4 presents the refinement rules to map spreadsheets into databases and in Section 5.5 we draw our conclusions.

5.2 Motivational Example

Throughout this chapter we will use a well-known example adapted from (Connolly and Begg 2001) and reproduced in the spreadsheet illustrated in Figure 5.2. This spreadsheet implements a property renting system, gathering information about renters, owners and rents. It also stores prices and dates of renting.



	A	B	C	D	E	F	G	H	I	J	K	L
1	**renterNr**	**propNr**	**renterNam**	**propAddress**	**country**	**rentStart**	**rentFinish**	**nrDays**	**rent**	**total**	**ownerNr**	**ownerNam**
2	cr76	pg4	John	6 Lawrence	UK	01-07-2000	31-08-2001	426	50	21300	co40	Tina
3	cr76	pg16	John	5 Nuvar Dr.	UK	01-09-2001	01-09-2002	365	70	25550	co93	Tony
4	cr56	pg4	Aline	6 Lawrence	UK	01-09-1999	10-06-2000	283	50	14150	co40	Tina
5	cr56	pg36	Aline	2 Manor Rd	UK	10-10-2000	01-09-2001	326	60	19560	co12	Anne
6	cr56	pg16	Aline	5 Nuvar Dr.	UK	01-11-2002	10-10-2003	343	70	24010	co93	Tony

Figure 5.2: A spreadsheet representing a property renting system.

The labels in the first row have the following meaning: **renterNr** represents a unique code for each renter, **propNr** a unique code for each property, **renterNam** the name of the renter, **propAddress** the address of each property, **country** the country of the property, **rentStart** and **rentFinish** the date of the beginning and ending of the renting, **nrDays** the number of renting days, **rent** the rent per day of a property, **total** the total amount to pay by the renting, **ownerNr** a unique code for the owner of a property and **ownerNm** the name of the owner.

Notice that column *nrDays* is defined by a formula, $= rentFinish - rentStart$, which is instantiated in the second row as `=G2-F2`. Column *total* is also defined by a formula, $= nrDays * rent$, which is instantiated in the second row as `=I2*H2`.

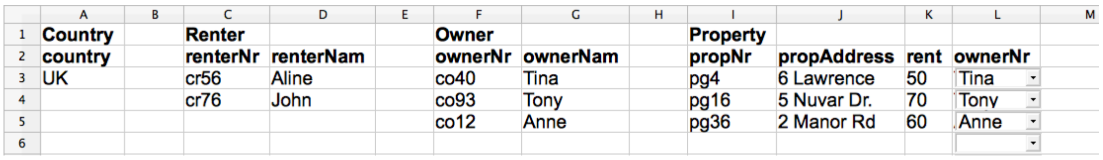
Like in the movie renting example presented in Chapter 4, this spreadsheet defines a valid model to represent the information of the renting system. However, it contains redundant information. For example, the displayed data specifies the house renting of two renters only, but their names are included 5 times. This kind of redundancy makes the maintenance and update of the spreadsheet complex and error-prone. A mistake is easily made, for example by mistyping a name and thus corrupting the data. As we explained before, the same information can be stored without redundancy.

Based on the data in our example spreadsheet and using the techniques presented in Chapter 3, we can discover the following relational database schema:

$$\begin{array}{ll}
Country & (\underline{country}) \\
Renter & (\underline{renterNr}, renterNam) \\
Owner & (\underline{ownerNr}, ownerNm) \\
Property & (\underline{propNr}, propAddress, rent, \#ownerNr) \\
<Renting> & (\underline{\#renterNr, \#propNr, \#country, rentStart, rentFinish}, nrDays, total)
\end{array}$$

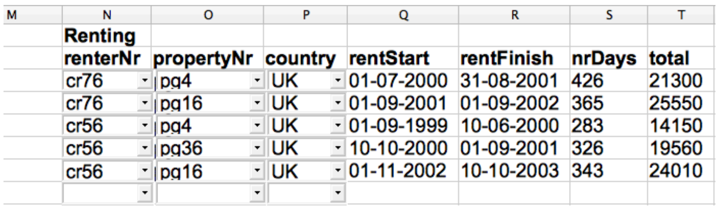
This schema defines the entities involved in the property renting system: namely property *countries*, *renters*, *properties* and their *owners* and the *renting* action itself.

From this schema we would like to, either store the data in a relational database management system, or create an improved spreadsheet. Figure 5.3 presents such an optimized and a modular spreadsheet for our running example. Notice that, these figures are from the same sheet, but for presentation reasons they are separated. In fact, all tables could be stored in separated sheets in a single workbook.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**Country**		**Renter**			**Owner**			**Property**				
2	**country**		**renterNr**	**renterNam**		**ownerNr**	**ownerNam**		**propNr**	**propAddress**	**rent**	**ownerNr**	
3	UK		cr56	Aline		co40	Tina		pg4	6 Lawrence	50	Tina	
4			cr76	John		co93	Tony		pg16	5 Nuvar Dr.	70	Tony	
5						co12	Anne		pg36	2 Manor Rd	60	Anne	
6													

(a) First part of the refactored properties spreadsheet.



M	N	O	P	Q	R	S	T
	Renting						
	renterNr	**propertyNr**	**country**	**rentStart**	**rentFinish**	**nrDays**	**total**
	cr76	pg4	UK	01-07-2000	31-08-2001	426	21300
	cr76	pg16	UK	01-09-2001	01-09-2002	365	25550
	cr56	pg4	UK	01-09-1999	10-06-2000	283	14150
	cr56	pg36	UK	10-10-2000	01-09-2001	326	19560
	cr56	pg16	UK	01-11-2002	10-10-2003	343	24010

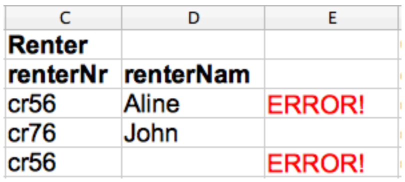
(b) Second part of the refactored properties spreadsheet.

Figure 5.3: The spreadsheet after applying the third normal form refactoring.

This new spreadsheet consists of five tables/modules (each one delimited by the empty column). In Figure 5.3a, from left to right, we have a table for *countries*, another for *renters*, for *owners* and for *properties*. Figure 5.3b contains the table for the *renting* action itself.

As we explained before, the obtained modularity solves three well-known problems in databases, namely the insertion, modification and deletion anomalies. These problems do not exist in the generated spreadsheet because data is normalized.

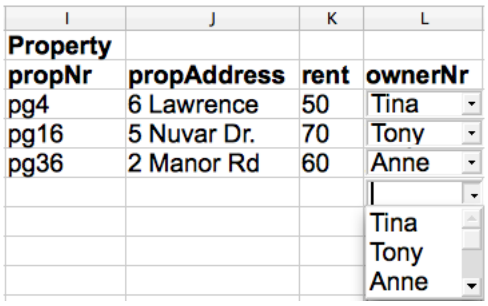
Moreover, we would like that the generated spreadsheet would respects the schema. For example, in the *renters* table, the generated spreadsheet would not allow the user to introduce two renters with the same code, that is, the same *renterNr*. If that error occurs, the spreadsheet system should warn the user as shown in Figure 5.4. Obviously, it is not possible to perform this validation in the original spreadsheet.



C	D	E
Renter		
renterNr	**renterNam**	
cr56	Aline	ERROR!
cr76	John	
cr56		ERROR!

Figure 5.4: If the user introduces a new row in a table with a previously used *renterNr* the spreadsheet will immediately produce an error.

The reader may have noticed that, for example, column L (for example, Figure 5.3) contains *combo boxes* on the cells. From the relational schema, we can see that this column, *ownerNr*, is a foreign key to the owner code in the *owners* table. In these cases, we would like that the new spreadsheet guarantees that the user only inserts values that already exist in the referenced table, since this is the definition of a foreign key. In fact, these columns should be locked for editing. This feature is illustrated in Figure 5.5.



I	J	K	L
Property			
propNr	**propAddress**	**rent**	**ownerNr**
pg4	6 Lawrence	50	Tina
pg16	5 Nuvar Dr.	70	Tony
pg36	2 Manor Rd	60	Anne
			Tina
			Tony
			Anne

Figure 5.5: Foreign key columns are filled in using combo boxes.

The refactored spreadsheet not only improves modularity and detects the introduction of incorrect data, but also eliminates redundancy: indeed, the redundancy present in the original spreadsheet has been eliminated. As expected, the names of the two renters occur only once. As we will demonstrate in the remaining sections of this chapter, the process of converting the data to the new format can be formalized and automated.

After establishing a mapping between the original spreadsheet and a relational database schema, we may want to use SQL (Date 1986) to query the spreadsheet. Regarding the properties renting information, one may want to know who are the renters of the properties that where rented between January, 2000 and January, 2002? Such queries are difficult to formulate in the spreadsheet environment. In SQL, the above question can be formulated as follows:

```
select renterNr from renting
   where rentStart between '01-01-00' and '01-01-02'
```

In the next sections, we will formalize the correspondence between spreadsheets and relational schemas using data refinement rules. Moreover, we will present a framework that implements the transformation rules. In fact, the example presented in this section was automatically processed by our framework.

5.3 A Constraint-aware Rewriting System

As we explained in the previous section, given the relational schema, we want to derive conversion functions to migrate data from a spreadsheet to a database with the synthesized schema, and vice versa. Thus, we are confronted with a *two level-transformation* problem, where a transformation at the level of types (schemas) is coupled with transformations at the level of values (data), as illustrated in Figure 5.1.

Such two-level transformations can be formalized as data refinements (Morgan and Gardiner 1990; Oliveira 1990, 2008) and can be supported with a term rewriting system (Lämmel and Visser 2003; Visser and Saraiva 2004; Visser 2005) where type representations are rewritten and conversion functions are synthesized at each rewrite step. For this purpose, we make use of the *Two-Level Transformation* (2LT) system[1] (Cunha *et al.* 2006; Cunha and Visser 2007a,b; Visser 2008; Alves *et al.* 2008). In this section, we provide the necessary background.

5.3.1 Data Refinements

Data refinement theory provides an algebraic framework for calculating with data types. Refining a data type A to a data type B can be captured by the following diagram:

$$\begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B \arrow[l, bend left, "from"] \end{tikzcd} \quad \text{where} \quad \begin{array}{l} to : A \rightarrow B \text{ is an injective function;} \\ from : B \rightarrow A \text{ a surjective function;} \\ from \circ to = id_A \text{ (identity function on } A); \end{array}$$

Notice that in general, to does not need to be a function, but can be a total relation (Morgan and Gardiner 1990; Oliveira 1990, 2008).

Refinements can be composed, that is,

$$\text{if } \begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B \arrow[l, bend left, "from"] \end{tikzcd} \text{ and } \begin{tikzcd} B \arrow[r, bend left, "to'"] \arrow[r, phantom, "\leqslant"] & C \arrow[l, bend left, "from'"] \end{tikzcd} \text{ then } \begin{tikzcd} A \arrow[r, bend left, "to' \circ to"] \arrow[r, phantom, "\leqslant"] & C \arrow[l, bend left, "from \circ from'"] \end{tikzcd}$$

Also, transformations in specific parts of a data type can be propagated to the global data type in which they are embedded, that is,

[1]This system is available at `http://code.google.com/p/2lt`.

$$\text{if } \begin{tikzcd} A \arrow[rr, bend left, "to"] \arrow[rr, phantom, "\leqslant"] & & B \arrow[ll, bend left, "from"] \end{tikzcd} \text{ then } \begin{tikzcd} \mathsf{F}A \arrow[rr, bend left, "\mathsf{F}\ to"] \arrow[rr, phantom, "\leqslant"] & & \mathsf{F}B \arrow[ll, bend left, "\mathsf{F}\ from"] \end{tikzcd}$$

where F is a functor that models the context of the transformation. A functor captures *i)* the embedding of local data types inside global data types and *ii)* the lifting of value-level functions to and $from$ on the local data types to value-level transformations on the global data types.

In the particular case where a refinement works in both directions we have an isomorphism $A \cong B$:

$$\text{if } \begin{tikzcd} A \arrow[rr, bend left, "to"] \arrow[rr, phantom, "\leqslant"] & & B \arrow[ll, bend left, "from"] \end{tikzcd} \text{ and } \begin{tikzcd} A \arrow[rr, bend left, "to"] \arrow[rr, phantom, "\geqslant"] & & B \arrow[ll, bend left, "from"] \end{tikzcd} \text{ then } \begin{tikzcd} A \arrow[rr, bend left, "to"] \arrow[rr, phantom, "\cong"] & & B \arrow[ll, bend left, "from"] \end{tikzcd}$$

A common example of a refinement (Cunha *et al.* 2006) is that maps from natural numbers to some type, $\mathbb{N} \rightharpoonup A$, are the implementation of lists of that type, $A^\star$:

$$\begin{tikzcd} A^\star \arrow[rr, bend left, "seq2index"] \arrow[rr, phantom, "\leqslant"] & & \mathbb{N} \rightharpoonup A \arrow[ll, bend left, "list"] \end{tikzcd}$$

The function *seq2index* creates a map (or finite function) where the keys are the indexes of the elements of the list. *list* collects the elements in the map. For example,

$$seq2index\,[\texttt{'a'},\texttt{'z'},\texttt{'x'},\texttt{'k'}] = \{1 \mapsto \texttt{'a'}, 2 \mapsto \texttt{'z'}, 3 \mapsto \texttt{'x'}, 4 \mapsto \texttt{'k'}\}$$
$$list\,\{1 \mapsto \texttt{'a'}, 2 \mapsto \texttt{'z'}, 3 \mapsto \texttt{'x'}, 4 \mapsto \texttt{'k'}\} = [\texttt{'a'},\texttt{'z'},\texttt{'x'},\texttt{'k'}]$$

Data Types with Constraints

A constraint on a data type can be modeled as a unary predicate, that is, a boolean function which distinguishes between legal values and values that violate the constraint. A constraint is associated to a type writing it as a subscript: A_ϕ, where $\phi : A \to \mathbb{B}$ is a total function (Oliveira 1998). For example, the following data type represents two relational tables with a foreign key constraint:

$$((A \rightharpoonup B) \times (A \rightharpoonup C))_{\delta\circ\pi_1 \subseteq \delta\circ\pi_2}$$

The projection functions $\pi_1 :: A \times B \to A$ and $\pi_2 :: A \times B \to B$ are used to select the left or right table and $\delta :: (A \rightharpoonup B) \to (Set\, A)$ to select the domain of a map. Additionally, a variant of the set inclusion operator lifted to point-free functions is also used:

$$\subseteq : (A \to Set\ B) \to (A \to Set\ B) \to (A \to \mathbb{B})$$

When a second constraint is added to a constrained data type, both constraints can be composed with logical conjunction:

$$(A_\phi)_\psi \equiv A_{\phi \wedge \psi}$$

When a constraint is present on a data type under a functor, the constraint can be pulled up through the functor (for a categorical proof, see (Oliveira 1998)):

$$\mathsf{F}(A_\phi) \equiv (\mathsf{F}A)_{(\mathsf{F}\phi)}$$

For example, a constraint on the elements of a list can be pulled up to a constraint on the list:

$$(A_\phi)^\star \equiv (A^\star)_{\phi^\star}$$

A concrete example for this type is as follows:

$$(Int_{>0})^\star \equiv (Int^\star)_{(>0)^\star}$$

This means that it is equivalent to allow only integers greater than zero to be part of a list (left part of the equation) and to have a list of all integers and then filter out the ones greater than zero (right part of the equation).

Data Type Refinement with Constraints

The laws of the data refinement calculus must be enhanced to deal with constrained data types. Firstly, if a constrained data type is refined with a "classic" law, *i.e.* a law that does not involve constraints, the constraint must be properly propagated through the refinement:

$$\text{if } \begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B \arrow[l, bend left, "from"] \end{tikzcd} \text{ then } \begin{tikzcd} A_\phi \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B_{\phi \circ from} \arrow[l, bend left, "from"] \end{tikzcd}$$

Thus, the constraint of the source data type is propagated to the target data type, where it is post-composed with the backward conversion function *from*. Such compositions can give rise to opportunities for point-free program transformation, as we will see further on.

Several refinement laws can be changed from inequations to isomorphisms by adding a constraint to the target type. For example, the law for sum elimination, $A+B \leqslant A?+B?$, can be enhanced as to $A+B \leqslant A?+B?_{(\varepsilon \circ \pi_1)\oplus(\varepsilon \circ \pi_2)}$. Notice the use of point-free variants of exclusive disjunction ($\oplus$) and a test for emptiness of an optional (ε).

When applying a law that introduces a constraint to a data type that already has a constraint, the new and existing constraints must be combined:

$$\begin{tikzcd}\text{if } A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B_\psi \arrow[l, bend left, "from"] \text{ then } A_\phi \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & (B_\psi)_{\phi \circ from} \arrow[l, bend left, "from"] \equiv B_{\psi \wedge (\phi \circ from)}\end{tikzcd}$$

This is the invariant pulling theorem of (Oliveira 1998). A more general case arises when not only the target, but also the source is constrained in the law that is applied:

$$\begin{tikzcd}\text{if } A_\chi \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B_\psi \arrow[l, bend left, "from"] \text{ and } \phi \Rightarrow \chi \text{ then } A_\phi \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & B_{\psi \wedge (\phi \circ from)} \arrow[l, bend left, "from"]\end{tikzcd}$$

Here it is used a point-free variant on logical implication ($\Rightarrow$) to state that the actual constraint ϕ on A must imply the required constraint χ.

In addition to introduction and propagation, constraints can also be weakened or even eliminated, by virtue of the following: if $\phi \Rightarrow \chi$ then $A_\phi \leqslant A_\chi$.

In the special case that ψ is the constant true predicate, such weakening boils down to elimination of a constraint.

5.3.2 Two-Level Transformations with Constraints

The data refinement theory presented in the previous section was implemented in HASKELL as a rewriting system named *two-level transformation* (2LT) (Cunha *et al.* 2006; Alves *et al.* 2008). We will now briefly introduce this system. A type-safe representation of types and functions is constructed using *Generalized Algebraic Data Types* (GADT) (P. Jones *et al.* 2004; Hinze *et al.* 2006), which allows to assign more precise types to data constructors by restricting the variables of the data type in the constructors' result types. To represent types, the following GADT is used:

```
data Type t where
  Int    :: Type Int                               -- representation of integer
  String :: Type String                            -- representation of string
  [·]    :: Type a → Type [a]                      -- representation of list
  Maybe  :: Type a → Type (Maybe a)                -- representation of optional
  · × ·  :: Type a → Type b → Type (a,b)           -- representation of product
  · ⇀ ·  :: Type a → Type b → Type (a ⇀ b)         -- representation of map
  a_φ    :: Type a → PF (Pred a) → Type a          -- representation of invariants
```

Notice that the HASKELL type `Map a b` for finite maps is rendered as $a \rightharpoonup b$.

The following type is used to encode the type constraints:

```
type Pred a = a → Bool
```

That is, *Pred a* is a shorthand for $a \to Bool$. The notation a_ϕ denotes a representation of type a, constrained by a boolean function represented by ϕ.

Functions are represented in a point-free style, that is, without any variables. Their representation is accomplished by the following GADT:

```
data PF a where
  id        :: PF (a → a)                                  -- identity function
  pnt       :: a → PF (One → a)                            -- constant
  π1        :: PF ((a,b) → a)                              -- left projection of a pair
  π2        :: PF ((a,b) → b)                              -- right projection of a pair
  list2set  :: PF ([a] → Set a)                            -- list to set
  ρ         :: PF ((a ⇀ b) → Set b)                        -- range of a map
  δ         :: PF ((a ⇀ b) → Set a)                        -- domain of a map
  CompList  :: PF ([(a,b)] → [(b,b)])                      -- composition of lists of pairs
  ListId    :: PF ([(a,b)] → [(b,b)])                      -- list of pairs
  ·⋆        :: PF (a → b) → PF ([a] → [b])                 -- list function map
  ·⋆        :: PF (a → b) → PF (Set a → Set b)             -- set function map
                                                           -- logical operator
  · ∧ ·     :: PF (Pred a) → PF (Pred a) → PF (Pred a)
                                                           -- product of functions
  · × ·     :: PF (a → b) → PF (c → d) → PF ((a,c) → (b,d))
```

$$
\begin{array}{ll}
 & \text{-- split of functions} \\
\cdot \triangle \cdot & :: PF\ (a \to b) \to PF\ (a \to c) \to PF\ (a \to (b,c)) \\
 & \text{-- composition of functions} \\
\cdot \circ \cdot & :: Type\ b \to PF\ (b \to c) \to PF\ (a \to b) \to PF\ (a \to c) \\
 & \text{-- set inclusion} \\
\cdot \subseteq_s \cdot & :: Type\ b \to PF\ (a \to Set\ b) \to PF\ (a \to Set\ b) \to PF\ (Pred\ a) \\
 & \text{-- list inclusion} \\
\cdot \subseteq_l \cdot & :: Type\ b \to PF\ (a \to Set\ b) \to PF\ (a \to Set\ b) \to PF\ (Pred\ a) \\
Sstable2table & :: PF\ ([(a,b)] \to (a \rightharpoonup b)) \qquad \text{-- SS table to RDB table} \\
Table2sstable & :: PF\ ((a \rightharpoonup b) \to [(a,b)]) \qquad \text{-- RDB table to SS table}
\end{array}
$$

This GADT represents the types of the functions used in the transformations. For example, π_1 represents the type of the function that project the first part of a pair. The comments should clarify which function each constructor represents.

Given these representations of types and functions, we can turn to the encoding of refinements. Each refinement is encoded as a two-level rewriting rule:

$$
\begin{array}{l}
\textbf{type}\ Rule = \forall\ a\ .\ Type\ a \to Maybe\ (View\ (Type\ a)) \\
\textbf{data}\ View\ a\ \textbf{where} \\
\quad View :: Rep\ a\ b \to Type\ b \to View\ (Type\ a) \\
\textbf{data}\ Rep\ a\ b = Rep\ \{to :: PF\ (a \to b), from :: PF\ (b \to a)\}
\end{array}
$$

Although the refinement is from a type a to a type b, this cannot be directly encoded since the type b is only known when the transformation completes, so the type b is represented as a *view* of the type a. A view means that a type a can be represented by a type b, denoted $Rep\ a\ b$, if both a function $to :: a \to b$ and an inverse function $from :: b \to a$ are given.

The following code implements a rule to transform a list into a map:

$$
\begin{array}{l}
listmap :: Rule \\
listmap\ ([a]) = Just\ (View\ (Rep\ \{to = seq2index, from = tolist\})\ (Int \rightharpoonup a)) \\
listmap\ \_ = mzero
\end{array}
$$

The witness functions have the following signatures (their code here is not important):

$$
\begin{array}{l}
tolist :: (Int \rightharpoonup a) \to [a] \\
seq2index :: [a] \to (Int \rightharpoonup a)
\end{array}
$$

This rule receives the type of a list of a, $[a]$, and returns a view over the type map of integers to a, $Int \rightharpoonup a$. The witness functions are returned in the representation *Rep*. If any other argument than a list if received, the rule fails, returning *mzero*. This combinator is the identity in a monad; for example, if we were working with lists, *mzero* would be the empty list. All the rules contemplate this case and so we will not show it in the other rules.

Given this encoding of individual rewrite rules, a complete rewrite system can be build via the following constructors:

$$
\begin{array}{lll}
nop & :: Rule & \text{-- identity} \\
\triangleright & :: Rule \rightarrow Rule \rightarrow Rule & \text{-- sequential composition} \\
\oslash & :: Rule \rightarrow Rule \rightarrow Rule & \text{-- left-biased choice} \\
many & :: Rule \rightarrow Rule & \text{-- repetition} \\
once & :: Rule \rightarrow Rule & \text{-- arbitrary depth rule application}
\end{array}
$$

Details on the implementation of these combinators can be found elsewhere (Cunha *et al.* 2006).

After a two-level rewriting system has been constructed and applied to a type a to arrive at a type b, the synthesized conversion function can be applied with the following convenience HASKELL functions:

$$
\begin{array}{l}
forth :: View\ (Type\ a) \rightarrow Type\ b \rightarrow a \rightarrow Maybe\ b \\
back :: View\ (Type\ a) \rightarrow Type\ b \rightarrow b \rightarrow Maybe\ a
\end{array}
$$

Thus, the *forth* function will perform the forward conversion to a value of type a, while *back* performs the backward conversion.

5.3.3 Representing Spreadsheets and Relational Databases

In our approach, spreadsheets are represented by a product of tables. A table is a list of rows and each row is a product of values. For example, the code shown next represents a spreadsheet with a single table containing three rows and three columns. Its type is $(String, (String, String))^{\star}$:

```
[("cr76",("John","5554434")),
 ("cr56",("Aline","5552122")),
 ("cr56",("Aline","5552122"))]
```

The reader may notice that this model does not directly represent the definition of spreadsheets given in Chapter 2. In this chapter we choose this type to specify spreadsheets because it facilitates the migration to and from databases.

To represent a relational database table we use a map from the key to the non-key attributes. A product of maps defines a database. The database representation of the above spreadsheet is shown next. Its type is $String \rightharpoonup (String, String)$:

```
{"cr76" ↦ ("John","5554434"),
 "cr56" ↦ ("Aline","5552122")}
```

We will see in the next section that these representations are not sufficient to ensure that no data is lost during the transformations. Thus, we will show how to use invariants to guarantee that no data is lost.

Correctly Representing Spreadsheets

It is possible to transform a spreadsheet table, $(A \times B)^{\star}$, into a relational table, $A \rightharpoonup B$ (see Section 5.4), but the spreadsheet type must have a constraint imposing that there exist a functional dependency between the elements in the column of type A and the column of type B (both A and B can be sets of columns). Thus, the type that we will use to represent spreadsheets is $(A \times B)^{\star}_{CompList \subseteq_l ListId_B}$. To better understand how the invariant works, we explain the meaning of the functions used to define it:

CompList Uses a list of pairs as a relation (in the mathematical sense) and composes its inverse with it. For example, suppose it receives the list $[(1,2),(3,6)]$. Its inverse is $[(2,1),(6,3)]$. The composition results in $[(2,2),(6,6)]$.

ListId Is the list resulting from transforming the identity (mathematical) relation into a list. For example, for integers it would be $[...,(1,1),(2,2),(3,3),(4,4),...]$.

$\cdot \subseteq_l \cdot$ Tests for list inclusion.

This definition of functional dependency is based on the one presented in (Oliveira 2005). The intuition behind this invariant is that if there is a functional dependency, *CompList* will produce a subset of the identity relation. Let us look at a couple of examples. Suppose we start with the middle list of data shown bellow. Its inverse is shown on the left and their composition on the right. As the reader may have noticed, the composition is a subset of the identity relation, and thus, the inclusion test will pass yielding that the list encodes a functional dependency.

$$\begin{array}{ccc}
 & & [(b1,b1), \\
[(b1,a1), & [(a1,b1), & (b1,b1), \\
(b1,a1), & (a1,b1), & (b1,b1), \\
(b2,a2)] & (a2,b2)] & (b1,b1), \\
 & & (b2,b2)]
\end{array}$$

On the other hand, starting with the relation shown bellow in the middle, its inverse on the left and their composition on the right, we can see that this composition is not a subset of the identity relation. Thus, the inclusion test will fail showing that the initial data does not encode a functional dependency.

$$\begin{array}{ccc}
 & & [(b1,b1), \\
[(b1,a1), & [(a1,b1), & (b1,b2), \\
(b2,a1), & (a1,b2), & (b2,b1), \\
(b3,a2)] & (a2,b3)] & (b2,b2), \\
 & & (b3,b3)]
\end{array}$$

From now on this invariant will be designated *fd*.

In the next section we will explain how to correctly represent relational databases. As we will see, the basic type shown before is not enough; some constraints on the type are necessary.

Correctly Representing Relational Databases

The most interesting rules are the ones that are isomorphisms since they allow us to move between both types without restrictions (apart from the ones imposed by the invariants). In fact, the type shown above for database tables, $A \rightharpoonup B$, is enough to guarantee that transformations form databases to spreadsheets do not lose any data. Unfortunately, this is not enough: we need to ensure the identity when doing a round trip, that is, if we apply a transformation from a spreadsheet to a database and back, we want to have the same spreadsheet we started with. With the current representation, if we have repeated rows in the spreadsheet, this repetition is lost, and thus, we need to improve the database type. In this case, we need to store in the database representation the number of times a certain row is repeated. To ensure we keep this information, we use the following type to represent a database: $A \rightharpoonup (B \times \mathbb{N})$. The $\mathbb{N}$ component encodes the number of times a row is repeated in the spreadsheet.

This improved type ensures that we do not lose data on a round trip, but now we cannot ensure that a transformation from a database to a spreadsheet is correct. Suppose we have the database table $\{$ `"a"` $\mapsto (1,0)\}$. We cannot represent zero times a tuple in a spreadsheet. Once more, we need a better type: we need to guarantee that $\mathbb{N}$ is always greater than zero. Thus, the type used to represent relational database tables in our setting is $A \rightharpoonup (B \times \mathbb{N}_{>0})$.

In the following section we will present a set of rules to transform a spreadsheet into a relational database. Since we designed rich types to represent both spreadsheets and databases, all the rules are isomorphisms, and thus, we can apply the inverse transformation without losing data.

5.4 Migration Rules

In this section we will describe several data refinements needed to transform a spreadsheet into a relational database and vice versa. We also present a strategy to combine these refinements in order to obtain a single rule that can transform any spreadsheet into a database and vice versa.

5.4.1 Refining a Spreadsheet Table to a Relational Table

The simplest rule is the one that transforms a single spreadsheet table into a relational table and vice versa. Given a table with data that respects a functional dependency, the forward transformation will produce a relational table. This forward transformation is accompanied by an inverse backward one. The following diagram represents this rule:

$$\begin{tikzcd}
(A \times B)^{\star}_{CompList \subseteq_l ListId_B} \arrow[r, bend left, "Sstable2table"] \arrow[r, phantom, "\cong"] & A \rightharpoonup (B \times \mathbb{N}_{>0}) \arrow[l, bend left, "Table2sstable"]
\end{tikzcd}$$

The transformation *Table2sstable*, that is, from the database to the spreadsheet, does not lose any information because each element of the map is transformed into a pair (an element of the list) with the key being transposed to the first element of the pair and the value (except the $\mathbb{N}$ component) to the second element of the pair. The $\mathbb{N}$ component will be represented in the number of times that row will appear repeated in the spreadsheet.

The other transformation, *Sstable2table*, from spreadsheets to databases is more dangerous in the sense that data could be lost. Suppose we have the list $[(1,2),(1,3)]$.

This is valid in the spreadsheet representation, but it cannot be represented in the database (at least directly). In fact, the invariant in the spreadsheet type prevents that the transformation is applied to cases like this. It could be applied if the 2 and the 3 were equal, which clearly is not the case. It could also be applied if the 1s were different, which is also not the case.

As we explained before, a round trip also does not lose any data. Thus, and since no information is lost in either of the transformations, this rule is an isomorphism.

The rule is a directly implemented in HASKELL as follows:

$$
\begin{array}{l}
\mathit{sstable2table} :: \mathit{Rule} \\
\mathit{sstable2table}\ [a \times b]_{\mathit{CompList} \subseteq_l \mathit{ListId}} = \mathit{return}\ (\mathit{View}\ \mathit{rep}\ (a \rightharpoonup (b \times \mathit{Int}_{>0}))) \\
\quad \textbf{where} \\
\qquad \mathit{rep} = \mathit{Rep}\ \{\mathit{to} = \mathit{Sstable2table}, \mathit{from} = \mathit{Table2sstable}\} \\
\mathit{sstable2table}\ \_ = \mathit{mzero}
\end{array}
$$

It receives a list of pairs, $[a \times b]$, with the functional dependency invariant ($\mathit{CompList} \subseteq_l \mathit{ListId}$), representing the spreadsheet data encoding a functional dependency. It returns a view of the map type, $a \rightharpoonup (b \times \mathit{Int}_{>0})$, representing the database, with the representation (*rep*) given by the two functions *Sstable2table* and *Table2sstable*. For all the other inputs, the function fails, returning *mzero*.

Let us use this rule to map the table with information about renters of our running example. For now, let us assume that we have the renters' data isolated, that is, in its own table. This will be discussed later on.

$$
\begin{array}{l}
*\mathit{ghci}{>}\ \mathit{sstable2table}\ [\mathit{renterNr} \times \mathit{renterNam}]_{\mathit{CompList} \subseteq_l \mathit{ListId}} \\
\mathit{Just}\ (\mathit{View}\ (\mathit{Rep} < \mathit{to} > <\!\mathit{from}\!>)\ (\mathit{renterNr} \rightharpoonup (\mathit{renterNam} \times \mathit{Int}_{>0})))
\end{array}
$$

The invariant $\mathit{CompList} \subseteq_l \mathit{ListId}$ guarantees that the functional dependency is present in the spreadsheet data type. It will be used later to guarantee that the data to migrate indeed encodes the functional dependency. Moreover, the returned *to* and *from* functions are the migration functions needed to map the data between the two models. To use the *to* and *from* function one can use the *forth* and *back* functions described in Section 5.3. Let us assume the following HASKELL definitions:

```
rentersData = [("cr76","John"),
               ("cr76","John"),
               ("cr56","Aline"),
               ("cr56","Aline"),
```

$(\texttt{"cr56"},\texttt{"Aline"})]$

$renterNr = \texttt{"renterNr"}$
$renterNam = \texttt{"renterNam"}$
$rentersType = [renterNr \times renterNam]_{CompList \subseteq_l ListId}$
$Just\ viewt = sstable2table\ rentersType$
$targetRDBSchema = renterNr \rightharpoonup (renterNam \times Int_{>0})$

Executing the forward transformation, we obtain the following result:

$*ghci\!>\! forth\ viewt\ targetRDBSchema\ rentersData$
$Just\ (fromList\ [(\texttt{"cr76"},(\texttt{"John"},2)),(\texttt{"cr76"},(\texttt{"John"},2)),$
$(\texttt{"cr56"},(\texttt{"Aline"},3)),(\texttt{"cr56"},(\texttt{"Aline"},3)),(\texttt{"cr56"},(\texttt{"Aline"},3))])$

The result can be a bit cryptic: since the *forth* function is partial and since in this case it did not fail, it returns *Just* the result. This is the way HASKELL has to express that the function succeeded returning something. Unfortunately, HASKELL does not have a very nice way of showing maps. It shows a list of pairs where the first component is the key of the map and the second component the value. This list is preceded by the function *fromList* which receives a list and returns a map. If we execute the result, we get the following new result:

$*ghci\!>\! fromList\ [(\texttt{"cr76"},(\texttt{"John"},2)),(\texttt{"cr76"},(\texttt{"John"},2)),$
$(\texttt{"cr56"},(\texttt{"Aline"},3)),(\texttt{"cr56"},(\texttt{"Aline"},3)),(\texttt{"cr56"},(\texttt{"Aline"},3))]$
$fromList\ [(\texttt{"cr56"},(\texttt{"Aline"},2)),(\texttt{"cr76"},(\texttt{"John"},3))]$

As the reader can see, it will show the map in the same format, but now without the repeated values, which is exactly what we want in the database. For a matter of presentation, we wrote a function that shows the maps in a more usual way. For our example it returns the following result:

$\{\texttt{"cr56"} \mapsto (\texttt{"Aline"},2),\texttt{"cr76"} \mapsto (\texttt{"John"},3)\}$

5.4.2 Refining Tables with Foreign Key in the Primary Key

A pair of spreadsheet tables where the primary key of the first table contains a foreign key to the primary key of the second table, can be refined to a pair of relational tables using the following law, termed *sstables2tables*:

$$\begin{tikzcd}
((A \times B)^{\star}_{fd} \times (C \times D)^{\star}_{fd})_{(\pi_{AE} \circ \pi_1)^{\star} \circ \pi_1 \subseteq (\pi_{CE} \circ \pi_1)^{\star} \circ \pi_2} \arrow[d, bend left, "Sstable2table \times Sstable2table"] \arrow[d, phantom, "\cong"] \\
((A \rightharpoonup (B \times \mathbb{N}_{>0})) \times (C \rightharpoonup (D \times \mathbb{N}_{>0})))_{\pi_{AE} \circ \delta \circ \pi_1 \subseteq \pi_{CE} \circ \delta \circ \pi_2} \arrow[u, bend left, "Table2sstable \times Table2sstable"]
\end{tikzcd}$$

The invariant guarantees the existence of a foreign key from the primary key of the first table to the primary key of the second one. The projection π_{AE} has type $\pi_{AE} : A \rightarrow E$ and π_{CE} the type $\pi_{CE} : C \rightarrow E$. The type A must be a tuple of the form $A_1 \times \ldots \times E \times \ldots \times A_n$ and C of the form $C_1 \times \ldots \times E \times \ldots \times C_m$. This allows that just part of the primary key of each table is used in the foreign key definition, but it also allows that the whole key is used.

When transforming the spreadsheet model into the relational schema, the invariant must be updated to work on the new type. Thus, instead of writing $(\pi_{AE} \circ \pi_1)^{\star}$ we write $\pi_{AE} \circ \delta$ which selects the values of the columns that are foreign keys in the spreadsheet and in the database, respectively. The selection of the referenced columns is analogous to this one. A particular instance of this refinement occurs when π_{AE} degenerates on the identity function. In this case all the attributes of the primary key of the first table, are foreign keys to part of the attributes of the primary key of the second table. The diagram of this rule is shown next:

$$\begin{tikzcd}
((A \times B)^{\star}_{fd} \times (C \times D)^{\star}_{fd})_{\pi_1^{\star} \circ \pi_1 \subseteq (\pi_{CE} \circ \pi_1)^{\star} \circ \pi_2} \arrow[d, bend left, "Sstable2table \times Sstable2table"] \arrow[d, phantom, "\cong"] \\
((A \rightharpoonup (B \times \mathbb{N}_{>0})) \times (C \rightharpoonup (D \times \mathbb{N}_{>0})))_{\delta \circ \pi_1 \subseteq \pi_{CE} \circ \delta \circ \pi_2} \arrow[u, bend left, "Table2sstable \times Table2sstable"]
\end{tikzcd}$$

In this case, $A = E$ and thus the projection π_1 in the first table is enough to get the necessary values. On the database side, the domain of the first table is enough get the correct values.

Another instance of this refinement is when π_{CE} degenerates on the identity function, that is, part of the attributes of the primary key of the first table reference all the attributes of the primary key of the second one. The diagram of this instance is shown next:

$$\begin{tikzcd}
((A \times B)^{\star}_{fd} \times (C \times D)^{\star}_{fd})_{(\pi_{AE}\circ\pi_1)^{\star}\circ\pi_1 \subseteq \pi_1^{\star}\circ\pi_2} \arrow[d, bend left, "Sstable2table\times Sstable2table"] \arrow[d, phantom, "\cong"] \\
((A \rightharpoonup (B \times \mathbb{N}_{>0})) \times (C \rightharpoonup (D \times \mathbb{N}_{>0})))_{\pi_{AE}\circ\delta\circ\pi_1 \subseteq \delta\circ\pi_2} \arrow[u, bend left, "Table2sstable\times Table2sstable"]
\end{tikzcd}$$

In this case, $C = E$ and once more, the projection π_1 and the domain are enough to get the correct values. An example of a situation like this can be found in our running example. Remember tables *Renting* and *Renter*:

Renter	$(\underline{renterNr}, renterNam)$
$<$*Renting*$>$	$(\underline{\#renterNr, \#propNr, \#country, rentStart, rentFinish}, nrDays, total)$

In this case, part of the key of *Renting*, namely *renterNr*, is a foreign key to the entire key of *Renter*, namely *renterNr*.

A final instance of this rule occurs when both π_{AE} and π_{CE} degenerate on the identity function, meaning that all the attributes of the primary key of the first table are foreign keys to all the attributes of the primary key of the second table. In this case, the refinement changes as we show next:

$$\begin{tikzcd}
((A \times B)^{\star}_{fd} \times (C \times D)^{\star}_{fd})_{\pi_1^{\star}\circ\pi_1 \subseteq \pi_1^{\star}\circ\pi_2} \arrow[d, bend left, "Sstable2table\times Sstable2table"] \arrow[d, phantom, "\cong"] \\
((A \rightharpoonup (B \times \mathbb{N}_{>0})) \times (C \rightharpoonup (D \times \mathbb{N}_{>0})))_{\delta\circ\pi_1 \subseteq \delta\circ\pi_2} \arrow[u, bend left, "Table2sstable\times Table2sstable"]
\end{tikzcd}$$

In this instance, $A = E = C$, and thus both π_A and π_C are not necessary.

Notice that, for each rule refining a pair $A \times B$ there exists a dual one refining the pair $B \times A$, with the appropriate invariant.

5.4.3 Refining Tables with Foreign Key in the Non-key Attributes

In the previous section we have introduced refinement rules to manipulate tables with foreign keys on the primary key. In this section we present another set of rules to deal with foreign keys on the non-key attributes. The diagram of the general rule, termed *sstables2tables′*, is presented next:

$$\begin{tikzcd}
((A \times B)^{\star}_{fd} \times (C \times D)^{\star}_{fd})_{(\pi_{BE}\circ\pi_2)^{\star}\circ\pi_1 \subseteq (\pi_{CE}\circ\pi_1)^{\star}\circ\pi_2} \arrow[d, bend left, "Sstable2table \times Sstable2table"] \arrow[d, phantom, "\cong"] \\
((A \rightharpoonup (B \times \mathbb{N}_{>0})) \times (C \rightharpoonup (D \times \mathbb{N}_{>0})))_{\pi_{BE}\circ\rho\circ\pi_1 \subseteq \pi_{CE}\circ\rho\circ\pi_2} \arrow[u, bend left, "Table2sstable \times Table2sstable"]
\end{tikzcd}$$

As in the previous rule, when transforming the spreadsheet into the database, the invariant must be updated to work on the new type. Thus, instead of writing $(\pi_{BE} \circ \pi_1)^{\star}$ we write $\pi_{BE} \circ \delta$ which selects the values of the columns that are foreign keys in the spreadsheet and in the database, respectively. The selection of the referenced columns is analogous.

Once more, the projection π_{BE} has type $\pi_{BE} : B \rightarrow E$ and π_{CE} the type $\pi_{CE} : C \rightarrow E$. The type B must be a tuple of the form $B_1 \times \ldots \times E \times \ldots \times B_n$ and C of the form $C_1 \times \ldots \times E \times \ldots \times C_m$.

As in the previous set of rules, this refinement has three particular cases:

- The first one occurs when π_{BE} degenerates in the identity function, that is, when all the non-key attributes are foreign keys. In this case, the projection π_{BE} disappears from the invariants.

- The second case occurs when π_{CE} degenerates in the identity function, that is, all the attributes in the primary key of the second table are referenced by some foreign key. In this case, π_{CE} is removed from the invariants.

  An example of a situation like this can be found in our running example. Remember tables *Owner* and *Property*:

  $$\begin{array}{ll} Owner & (\underline{ownerNr}, ownerNm) \\ Property & (\underline{propNr}, propAddress, rent, \#ownerNr) \end{array}$$

  In this case, the entire key of *Owner*, namely *ownerNr*, is referenced by part of the non-key attributes of the table *Property*, namely *ownerNr*.

- The third and final case occurs when both π_{BE} and π_{CE} degenerate on the identity function. As expected, both projection functions disappear from the invariants.

Notice also that, in the three cases the types of the spreadsheet and database do not suffer any change to the original types. Only the invariants need to be updated as explained before.

5.4.4 Data Refinements as a Strategic Rewrite System

The individual refinement rules presented in the previous sections can be combined into compound rules and in a full transformational system using the strategy combinators (Lämmel and Visser 2003; Visser and Saraiva 2004; Visser 2005) as shown in Section 5.3.2. In particular, we define a compound rule to map a spreadsheet into a relational database:

```
$ss2rdb :: Rule$
$ss2rdb = simplifyInv \triangleright$
        $(many\ ((aux\ sstables2tables) \triangleright (aux\ sstables2tables'))) \triangleright$
        $(many\ (aux\ sstable2table))$
  **where**
    $aux :: Rule \rightarrow Rule$
    $aux\ r = ((once\ r) \triangleright simplifyInv) \triangleright ((many\ (once\ r)) \triangleright simplifyInv)$
```

This rule starts by simplifying the invariants through the rule *simplifyInv*. After that, the rules *sstables2tables* (Section 5.4.2) and *sstables2tables*$'$ (Section 5.4.3) are applied using the auxiliary rule *aux* which applies a rule and then simplifies the invariants repeatedly until it fails. In a final step, the remaining tables are transformed using the *sstable2table* (Section 5.4.1) rule.

This strategy requires the simplification of the invariants because pattern matching is performed not only on the type representations, but also on the invariants.

Another strategy is to transform directly each spreadsheet table into a relational table:

```
$direct\_ss2rdb :: Rule$
$direct\_ss2rdb = many\ (once\ sstable2table)$
```

This rule applies the transformation exhaustively until all the tables are migrated to the relational model.

Remember that, since our rules are isomorphism, they can be used in both directions.

5.5 Conclusions

In this chapter, we have shown how a bidirectional mapping can be established between a spreadsheet and an equivalent relational database. We have provided the formal foundations for such a mapping.

In Chapter 8 we will use this basis to enhanced the HAEXCEL framework to transform a tabular schema into a relational schema and on the fly to derive conversion functions between these schemas. Furthermore, our techniques also migrate the formulas between the models. Moreover, we have connected importers and exporters for SQL and spreadsheet formats to the HAEXCEL framework.

In particular, we have made the following contributions.

- We have extended the 2LT framework with new constraint-aware two-level refinements;
- We have defined conversion rules between tabular and relational data structures. Moreover, we have shown how these rules can be combined into a strategic rewrite system;
- We have combined this rewrite system with methods for discovering relational schemas in spreadsheets as explained in Chapter 3;
- Finally, we have shown how the resulting system can be employed to convert spreadsheets to relational databases and back. This allows refactoring of spreadsheets to reduce data redundancy and improve error detection and migration of spreadsheet applications.

Notwithstanding these contributions, our approach presents a number of limitations that we hope to remove in future. For example, we are currently only supporting a set of commonly used formulas, which remains to be enlarged to a wider range.

We did not present the formal proofs of the refinement rules. Nevertheless, these rules are simple, well designed and well tested in several spreadsheets/databases.

In the two-level rewrite system, syntactic matching is performed on representations of constraints. Such syntactic matching could be generalized to verification of logical implication of the actual constraint and the required constraint.

With respect to formulas and queries, we are not yet exploiting some interesting opportunities. For example, a formula or query expressed in terms of the source spreadsheet may be composed with the backward conversion function to obtain a query on the target database or refactored spreadsheet. The migration of queries has been explored in (Cunha and Visser 2007a) in the context of XML and SQL.

Chapter 6

Safe Evolution of Spreadsheets

Summary

To help avoid the introduction of errors when changing spreadsheets, models that capture the structure and interdependencies of spreadsheets at a conceptual level have been proposed. Thus, spreadsheet evolution can be made safe within the confines of a model.

As in any other model/instance setting, evolution may not only require changes at the instance level but also at the model level. When model changes are required, the safety of instance evolution cannot be guarded by the model alone.

In this chapter we design an appropriate representation of spreadsheet models, including the fundamental notions of formulæ and references. For these models and their instances, we have designed coupled transformation rules that cover specific spreadsheet evolution steps, such as the insertion of columns in all occurrences of a repeated block of cells. Each model-level transformation rule is coupled with instance-level migration rules from the source to the target model and vice versa. These coupled rules can be composed to create compound transformations at the model level inducing compound transformations at the instance level. This approach guarantees safe evolution of spreadsheets even when models change.

6.1 Introduction

In order to improve spreadsheet end-users' productivity, several techniques have been recently proposed, which guide end users to safely/correctly edit spreadsheets, like, for example, the use of spreadsheet templates (Abraham *et al.* 2005), *ClassSheets* (Engels and Erwig 2005; Cunha *et al.* 2010), and the inclusion of visual objects to provide editing assistance in spreadsheets (Cunha *et al.* 2009b). All these approaches propose a form of end user model-driven software development: a spreadsheet business model is defined, from which a customized spreadsheet application is generated guaranteeing the consistency of the spreadsheet data with the underlying model. In Chapter 7, we show an empirical study demonstrating that the use of model-based spreadsheets improves, in some conditions, end users' productivity.

Despite of its huge benefits, model-driven software development is sometimes difficult to realize in practice due to two main reasons: first, as some studies suggest, defining the business model of a spreadsheet can be a complex task for end users (Abraham and Erwig 2006a). As a result, they are unable to follow this spreadsheet development discipline. Second, things get even more complex when the spreadsheet model needs to be updated due to new requirements of the business model. End users need not only to evolve the model, but also to migrate the spreadsheet data so that it remains consistent with the model. To address the first problem, in Chapter 3 we have proposed a technique to derive the spreadsheet's business model, represented as a *ClassSheet*, from the spreadsheet data. In this chapter we address the second problem, that is, the co-evolution of the spreadsheet model and the spreadsheet data (that is, the instance of the model). Co-evolution of models and instances are supported by the two-level coupled transformation framework (Cunha *et al.* 2006).

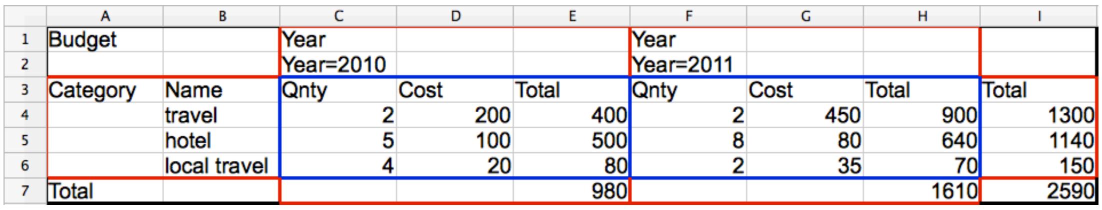
In this chapter we present an appropriate representation of a spreadsheet model, based on the *ClassSheet* business model, including the fundamental notions of formulæ, references, and expandable blocks of cells. The representation we present in this chapter is different from the one presented in the previous one. For the migration to databases, the previous model, defined as lists of tuples, was ideal since it had a nice match with the representation of databases (maps). In the evolution scenario, this is not sufficient, though. For example, if the evolution also changes formulas, a more advanced model is necessary.

For this model and its instance, we design coupled transformation rules that cover specific spreadsheet evolution steps, such as extraction of a block of cells into a separate sheet or insertion of columns in all occurrences of a repeated block of cells. Each model-level transformation rule is coupled with instance level migration rules from the source to the target model and vice versa. Moreover, these coupled rules can be composed to create compound transformations at the model level that induce compound transformations at the instance level. With this approach, spreadsheet evolution can be made type-safe, also when model changes are involved. To make these techniques available, we have implemented them in the HAEXCEL framework as will be described in Chapter 8.

This chapter is organized as follows. In Section 6.2 we discuss spreadsheet refactoring as our motivational example. In Section 6.3 we describe the framework to model and manipulate spreadsheets. Section 6.4 defines the rules to perform the evolution of spreadsheets and Section 6.5 concludes the chapter.

6.2 Motivational Example

Suppose a researcher's yearly budget for travel and accommodation expenses is kept in the spreadsheet shown in Figure 6.1 taken from (Engels and Erwig 2005).



	A	B	C	D	E	F	G	H	I
1	Budget		Year			Year			
2			Year=2010			Year=2011			
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total
4		travel	2	200	400	2	450	900	1300
5		hotel	5	100	500	8	80	640	1140
6		local travel	4	20	80	2	35	70	150
7	Total				980			1610	2590

Figure 6.1: Budget spreadsheet instance.

Note that throughout the years, cost and quantity are registered for three types of expenses: travel, hotel and local transportation. Formulas are used to calculate the total expense for each type in each year as well as the total expense in each year. Finally, a grand total is calculated over all years, both per type of expense and overall.

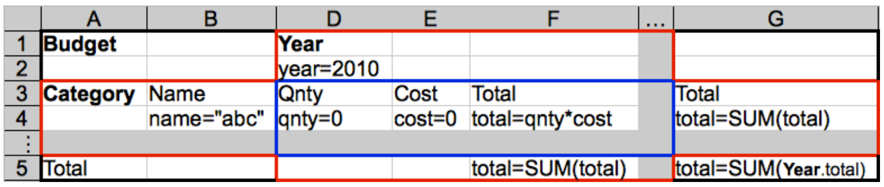
At the end of 2010, this spreadsheet needs to be modified to accommodate 2011 data (already in it). A novice spreadsheet user would typically take four steps to perform the necessary modifications:

- insert three new columns;
- copy all the labels;
- copy all the formulas (at least two);

- update all the necessary formulas in the last column.

A more advanced user would shortcut these steps by copy-inserting the 3-column block of 2010 and changing the label “2010” to “2011” in the copied block. If the insertion is done behind the last year, the range of the multi-year totals columns must be extended to include the new year. Apart from these two strategies, a mixed strategy may be employed. In any case, a conceptually unitary modification (*add year*) needs to be executed by an error-prone combination of steps.

As presented in Chapter 3, Erwig *et al.* have introduced *ClassSheets* as models of spreadsheets that allow spreadsheet modifications to be performed at the right conceptual level. For example, the *ClassSheet* in Figure 6.2 provides a model of our budget spreadsheet.



	A	B	D	E	F	...	G
1	**Budget**		**Year**				
2			year=2010				
3	**Category**	Name	Qnty	Cost	Total		Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost		total=SUM(total)
⋮							
5	Total				total=SUM(total)		total=SUM(Year.total)

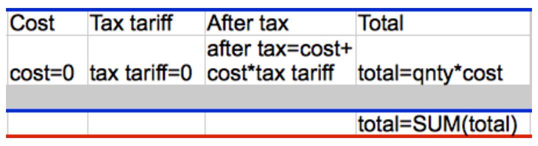
Figure 6.2: Budget spreadsheet model.

Before we present our techniques, let us briefly recall the definition of the *ClassSheets*. In this model, the repetition of a block of columns for each year is captured by the gray column labeled with the ellipsis. The horizontal repetition is marked in a analogous way. This makes it possible *(i)* to check whether the spreadsheet after modification still instantiates the same model, and *(ii)* to offer the user an unitary operation. Apart from (horizontal) block repetitions that support the extension with more years, this model features (vertical) row repetitions that support the extension with new expense types.

Unfortunately, situations may occur in which the model itself needs to be modified. For example, if the researcher needs to report expenses before and after tax, additional columns need to be inserted in the block of each year. Figure 6.3 shows the new spreadsheet as well as the new model that it instantiates.

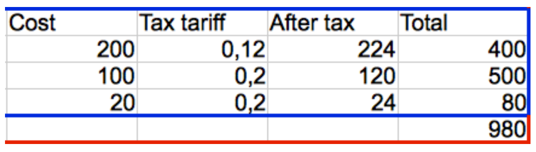
Note that, a modification of the year block in the model (inserting various columns) captures modifications to all repetitions of the block throughout the instance.

In this chapter, we will demonstrate that modifications to spreadsheet models can be supported by an appropriate combinator language, and that these model modifications can be propagated automatically to the spreadsheets that instantiate the models.



Cost	Tax tariff	After tax	Total
cost=0	tax tariff=0	after tax=cost+ cost*tax tariff	total=qnty*cost
			total=SUM(total)

(a) New budget model.



Cost	Tax tariff	After tax	Total
200	0,12	224	400
100	0,2	120	500
20	0,2	24	80
			980

(b) New budget instance.

Figure 6.3: New spreadsheet and the model that it instantiates.

In case of the budget example, the model modification is captured by the following expression:

```
addTax = once (
  inside "Year" (before "Total" (
    insertCol "Tax Tariff" ▷ insertCol "After tax"
  )))
```

The actual column insertions are done by the innermost sequence of two *insertCol* steps. The *before* and *inside* combinators specify the location constraints of applying these steps. The *once* combinator traverses the spreadsheet model to search for a single location where these constraints are satisfied and the insertions can be performed.

Applying the *addTax* transformation to the initial model (Figure 6.2) will yield:

- The modified model (Figure 6.3a);
- A spreadsheet migration function that can be applied to instances of the initial model (Figure 6.1) to produce instances of the new model (Figure 6.3b);
- An inverse spreadsheet migration function to backport instances of the modified model to instances of the initial model;

In the remainder of this chapter, we will explain the machinery required for this type of coupled transformation of spreadsheet instances and models. As models, we will use a variation on *ClassSheets* where references are modeled by projection functions. Model transformations propagate references by composing instance-level transformations with these projection functions.

6.3 A Framework for Evolution of Spreadsheets

Our approach to safe evolution of spreadsheets is, once more, based on data refinement theory, which provides an algebraic framework for calculating with data types and corresponding values (Morgan and Gardiner 1990; Oliveira 1990, 2008). As explained in the previous chapter, it consists of type-level coupled with value-level transformations. The type-level transformations deal with the evolution of the model and the value-level transformations deal with the instances of the model. Figure 6.4 depicts the general scenario of a transformation in this framework.

$$\begin{tikzcd} A \arrow[r, bend left, "to"] \arrow[r, phantom, "\leqslant"] & A' \arrow[l, bend left, "from"] \end{tikzcd}$$

A, A'	data type and transformed data type
to	witness function of type $A \to A'$ (injective)
from	witness function of type $A' \to A$ (surjective)

Figure 6.4: Coupled transformation of data type A into data type A'.

Remember that, each transformation is coupled with witness functions to and $from$, which are responsible for converting values of type A into type A' and back.

The 2LT framework is an HASKELL implementation of this theory (Cunha *et al.* 2006; Cunha and Visser 2007a,b; Visser 2008; Alves *et al.* 2008). It provides the basic combinators to define and compose transformations for data types and witness functions. Since 2LT is statically typed, transformations are guaranteed to be type-safe ensuring consistency of data types and data instances. For more details the reader is referred to Section 5.3 of this thesis.

6.3.1 *ClassSheets* and Spreadsheets in HASKELL

The 2LT was originally designed to work with algebraic data types. However, this representation is not expressive enough to represent *ClassSheet* specifications or their spreadsheet instances. To overcome this issue, we extended the 2LT representation presented on the previous chapter so it could support *ClassSheet* models, by introducing the following constructors in the existing GADT representation presented in Section 5.3.2:

```
data Type a where
  ...
```

$Value$	$:: Value \to Type\ Value$	-- plain value
		-- references
Ref	$:: Type\ b \to PF\ (a \to RefCell) \to PF\ (a \to b) \to Type\ a \to Type\ a$	
$RefCell$	$:: Type\ RefCell$	-- reference cell
$Formula$	$:: Formula \to Type\ Formula$	-- formulas
$LabelB$	$:: String \to Type\ LabelB$	-- block label
$\cdot = \cdot$	$:: Type\ a \to Type\ b \to Type\ (a,b)$	-- attributes
$\cdot \mid \cdot$	$:: Type\ a \to Type\ b \to Type\ (a,b)$	-- block horizontal composition
$\cdot\ \hat{}\ \cdot$	$:: Type\ a \to Type\ b \to Type\ (a,b)$	-- block vertical composition
$EmptyB$	$:: Type\ EmptyB$	-- empty block
$\underline{\cdot}$	$:: String \to Type\ HorH$	-- horizontal class label
$\mid \cdot$	$:: String \to Type\ VerV$	-- vertical class label
$\mid \underline{\cdot}$	$:: String \to Type\ Square$	-- square class label
$LabRel$	$:: String \to Type\ LabS$	-- relation class
$\cdot : \cdot$	$:: Type\ a \to Type\ b \to Type\ (a,b)$	-- labeled class
$\cdot : (\cdot)^{\downarrow}$	$:: Type\ a \to Type\ b \to Type\ (a,[b])$	-- labeled expandable class
$\cdot\ \hat{}\ \cdot$	$:: Type\ a \to Type\ b \to Type\ (a,b)$	-- class vertical composition
$SheetC$	$:: Type\ a \to Type\ (SheetC\ a)$	-- sheet class
$\cdot^{\to}$	$:: Type\ a \to Type\ [a]$	-- sheet expandable class
$\cdot \mid \cdot$	$:: Type\ a \to Type\ b \to Type\ (a,b)$	-- sheet horizontal composition
$EmptyS$	$:: Type\ EmptyS$	-- empty sheet

The comments should clarify what the constructors represent. Remember that, the values of type *Type a* are representations of type *a*. For example, if *t* is of type *Type Value*, then *t* represents the type *Value*. The following types are needed to construct values of type *Type a*:

data *EmptyBlock*	-- empty block
data *EmptySheet*	-- empty sheet
type $LabelB = String$	-- label
data $RefCell = RefCell1$	-- referenced cell
type $LabS = String$	-- square label
type $HorH = String$	-- horizontal label
type $VerV = String$	-- vertical label
data $SheetC\ a = SheetCC\ a$	-- sheet class

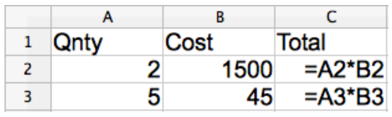
```
data SheetCE a = SheetCEC a                        -- expandable sheet class
                                                                  -- values
data Value = VInt Int | VString String | VBool Bool | VDouble Double
                                                                -- formulas
data Formula1 = FValue Value | FRef | FFormula String [Formula1]
```

Once more, the comments should clarify what each type represents.

To explain this representation we will use as an example a reduced version of the budget model presented in Figure 6.2. For this reduced model only three columns are defined: *quantity*, *cost per unit* and *total cost* (product of *quantity* by *cost per unit*).

```
purchase =
   | Price_List : Qnty ı Cost ı Total^
   | PriceList : (qnty = 0 ı cost = 0 ı total = FFormula × [FRef,FRef])↓
```

This *ClassSheet* specifies a class called *Price_List* composed by two parts vertically composed as indicated by the ^ operator. The first part is defined in the first row and defines the labels for three columns: *Qnty*, *Cost* and *Total*. The second row defines the rest of the class containing the definition of the three columns. The first two columns have as default value 0 and the third is defined by a formula (explained latter on). Note that, this part is vertically expandable, that is, it can be vertically repeated. In a spreadsheet instance this corresponds to the possibility of adding new rows. Figure 6.5 represents a spreadsheet instance of this model.



	A	B	C
1	Qnty	Cost	Total
2	2	1500	=A2*B2
3	5	45	=A3*B3

Figure 6.5: Spreadsheet instance of the *purchase ClassSheet*.

Note that in the definition of *Type a*, the constructors combining parts of the spreadsheet return pairs (for example, the operators to vertical and horizontally combine blocks, ·^· and · ı ·, return a pair each). Thus, a spreadsheet instance is written as nested pairs of values. The spreadsheet illustrated in Figure 6.5 is encoded in HASKELL as follows:

```
((Qnty, (Cost ,Total                    )),
[(2    ,(1500,FormulaFF × [FRef,FRef])),
 (5    ,(45   ,FormulaFF × [FRef,FRef]))])
```

The HASKELL type checker statically ensures that the pairs are well formed and are constructed in the correct order.

6.3.2 Specifying Formulas

Having defined a GADT to represent *ClassSheet* models, we now need a mechanism to define spreadsheet formulas. The safer way to specify formulas is making them strongly typed. Figure 6.6 depicts the scenario of a transformation with references. A reference from a cell s to the a cell t is defined using a pair of projections, *source* and *target*. These projections are statically-typed functions traversing the data type A to identify the cell defining the reference (s), and the cell to which the reference is pointing to (t). In this approach, not only the references are statically typed, but also always guaranteed to exist, that is, one cannot create a reference from/to a cell that does not exist.

$$\begin{tikzcd}
 & s & \\
A \arrow[ur, bend left, "source"] \arrow[dr, bend right, "target"'] \arrow[rr, Rightarrow, "T" description] \arrow[rr, bend left=20, "to"] & & A' \arrow[ul, bend right, "source'"'] \arrow[dl, bend left, "target'"] \arrow[ll, bend left=20, "from"] \\
 & t &
\end{tikzcd}$$

source	Projection over A identifying the reference
target	Projection over A identifying the referenced cell

$source' = source \circ from$

$target' = target \circ from$

Figure 6.6: Coupled transformation of data type A into data type A' with references.

The projections defining the reference and the referenced type, in the transformed type A', are obtained by post-composing the projections with the witness function *from*. When $source'$ and $target'$ are normalized they work on A' directly rather than via A. The formula specification, as previously shown, is specified directly in the GADT. However, the references are defined separately by defining projections over the data type. This is required to allow any reference to access any part of the GADT.

Using the spreadsheet illustrated in Figure 6.5, an instance of a reference from the formula *total* to *cost* is defined as follows:

$$\begin{array}{l} purchaseWithReference = \\ \quad Ref\ Int\ (fhead \circ head \circ (\pi_2 \circ \pi_2)^{\star} \circ \pi_2)\ (head \circ (\pi_1 \circ \pi_2)^{\star} \circ \pi_2)\ purchase \end{array}$$

Remember that, the second argument of *Ref* is the source (reference cell) and that the third is the target (referenced cell).

The *source* function refers to the first *FRef* in the HASKELL code shown after Figure 6.5. The *target* projection defines the cell it is pointing to, that is, it defines a reference to the the value 1500 in column *Cost*. Since the use of GADTs requires the definition of models combining elements in a pairwise fashion, it is necessary to descend into the structure using π_1 and π_2. The operator $\cdot^{\star}$ (*map* in functional programming) applies a function to all the element of a list and $fhead$ gets the first reference in a list of references.

Note that, our reference type has enough information about the cells and so we do not need value-level functions, that is, we do not need to specify the projection functions themselves, just their types. In the cases we reference a list of values, for example, constructed by the class expandable operator, we need to be specific about the element within the list we are referencing. For these cases, we use the type-level *head* (first element of a list) and $tail$ (all but first) to get the intended value in the list.

6.3.3 Representing Functions

At this point we are able to represent *ClassSheet* models, including formulas. In this section we discuss the definition of the witness functions *from* and *to*. Once again we rely on the definition of the GADT defined in Section 5.3.2:

```
data PF a where
  id     :: PF (a → a)                                  -- identity function
  π1     :: PF ((a,b) → a)                              -- left projection of a pair
  π2     :: PF ((a,b) → b)                              -- right projection of a pair
  pnt    :: a → PF (One → a)                            -- constant
                                                        -- split of functions
  ·△·    :: PF (a → b) → PF (a → c) → PF (a → (b,c))
                                                        -- product of functions
  ·×·    :: PF (a → b) → PF (c → d) → PF ((a,c) → (b,d))
                                                        -- composition of functions
  ·∘·    :: Type b → PF (b → c) → PF (a → b) → PF (a → c)
  ·⋆     :: PF (a → b) → PF ([a] → [b])                 -- map of functions
  head   :: PF ([a] → a)                                -- head of a list
  tail   :: PF ([a] → [a])                              -- tail of a list
  fhead  :: PF (Formula1 → RefCell)     -- head of the arguments of a formula
  ftail  :: PF (Formula1 → Formula1)    -- tail of the arguments of a formula
```

This GADT represents the types of the functions used in the transformations. For example, π_1 represents the type of the function that projects the first part of a pair. The comments should clarify which function each constructor represents.

6.4 Spreadsheet Evolution

In this section we define rules to perform spreadsheet evolution, which can be divided in three main categories: *Combinators*, used as helper rules, *Semantic* rules, intended to change the model itself (for example, add a new column), and *Layout* rules, designed to change the visual arrangement of the spreadsheet (for example, swap two columns).

6.4.1 Combinators

The semantic and the layout rules are defined to work on a specific part of the data type. The combinators defined next are then used to apply those rules in the desired places.

Pull Up All the References To avoid having references in different levels of the models, all the rules pull all the references to the topmost level of the model. To pull a reference in a particular place we use the following rule (we show just its first case):

$$\begin{aligned}&\mathit{pullUpRef} :: \mathit{Rule}\\&\mathit{pullUpRef}\ ((\mathit{Ref}\ \mathit{tb}\ \mathit{fRef}\ \mathit{tRef}\ \mathit{ta}) \mid \mathit{b2}) =\\&\quad \mathit{return}\ (\mathit{View}\ \mathit{idrep}\ (\mathit{Ref}\ \mathit{tb}\ (\mathit{fRef} \circ \pi_1)\ (\mathit{tRef} \circ \pi_1)\ (\mathit{ta} \mid \mathit{b2})))\end{aligned}$$

The representation *idrep* has the identity (*id*) function in both directions. If part of the model (in this case the left part of a horizontal composition) of a given type has a reference, it is pulled to the top level. This is achieved by composing the existing projections with the necessary functions, in this case π_1. This rule has two cases (left and right hand side) for each binary constructor (for example, horizontal/vertical composition).

To pull up all the references in all levels of a model we use the rule $\mathit{pullUpAllRefs} = \mathit{many}\ (\mathit{once}\ \mathit{pullUpRef})$. The *once* operator applies the *pullUpRef* rule somewhere in the type and the *many* ensures that this is applied everywhere in the whole model.

Apply After and Friends The combinator *after* finds the correct place to apply the argument rule (second argument) by comparing the given string (first argument) with the existing labels in the model. When it finds the intended place, it applies the rule to it. This works because our rules always do their task on the right-hand side of a type.

```
after :: String → Rule → Rule
after label r (label′ ı a) | label ≡ label′ = do
   View s l′ ← r label′
   return (View (Rep {to = to s × id, from = from s × id}) (l′ ı a))
```

Note that this definition is only part of the complete version since it only contemplates the case for horizontal composition of blocks $(\cdot \shortmid \cdot)$.

Other combinators were also developed, namely, *before*, *bellow*, *above*, *inside* and *at*. Their implementations are not shown since they are similar to the *after* combinator.

6.4.2 Semantic Rules

In this section we present rules that change the semantics of the model like, for example, by adding columns to a model.

Insert a Block One of the most fundamental rules is the insertion of a new block into a spreadsheet, formally defined as follows:

$$\begin{tikzcd} Block \arrow[r, bend left, "id \triangle (pnt\ a)"] \arrow[r, phantom, "\leqslant"] & Block \shortmid Block \arrow[l, bend left, "\pi_1"] \end{tikzcd}$$

This diagram means that a horizontal composition of two blocks refines a block when witnessed by two functions, *to* and *from*. The *to* function, $id \triangle (pnt\ a)$, is a split: it injects the existing block in the first part of the result without modifications (*id*) and injects the given block instance *a* into the second part of the result. The *from* function is π_1 since it is the one that allows the recovery of the previously existent block. The HASKELL version of the rule is presented next.

```
insertBlock :: Type a → a → Rule
insertBlock ta a tx | isBlock ta ∧ isBlock tx = do
   let rep = Rep {to = (id△(pnt a)), from = π₁}
```

```
    $View\ s\ t \leftarrow pullUpAllRefs\ (tx \shortmid ta)$
    $return\ (View\ (comprep\ rep\ s)\ t)$
```

The function *comprep* composes two representations. This rule receives the type of the new block *ta*, its default instance *a*, and returns a *Rule*. The returned rule is itself a function that receives the block to modify *tx* and returns a view of the new type. The first step is to verify if the given types are blocks, using the function *isBlock*. The second step is to create the representation *rep* with the witness functions given in the above diagram. Then, the references are pulled up in result type $tx \shortmid ta$. This returns a new representation *s* and a new type *t* (in fact, the type is the same $t = tx \shortmid ta$). The result view has as representation the composition of the two previous representations, *rep* and *s*, and the corresponding type *t*.

Rules to insert classes and sheets were also defined, but since these rules are similar to the rule for inserting blocks, we omit them.

Insert a Column To insert a column in a spreadsheet, that is, a cell with a label *lbl* and the cell bellow with a default value *df* and vertically expandable, we first need to create a new class representing it: $clas = \mid lbl : lbl\hat{}(lbl = df^{\downarrow})$. The label is used to create the default value $(lbl, [\,])$. Note that, since we want to create an expandable class, the second part of the pair must be a list. The final step is to apply *insertSheet*:

```
$insertCol :: String \rightarrow Formula1 \rightarrow Rule$
$insertCol\ lbl\ df@(FFormula\ name\ fs)\ tx \mid isSheet\ tx = \textbf{do}$
  $\textbf{let}\ clas = \mid lbl : lbl\hat{}(lbl = df^{\downarrow})$
  $((insertSheet\ clas\ (lbl, [\,])) \rhd pullUpAllRefs)\ tx$
```

Note the use of the rule *pullUpAllRefs* as explained before. The case shown in the above definition is for a formula as default value and it is similar to the value case. The case with a reference is more interesting and is shown next:

```
$insertCol\ lbl\ FRef\ tx \mid isSheet\ tx = \textbf{do}$
  $\textbf{let}\ clas = \mid lbl : Ref\ \perp\ \perp\ \perp\ (lbl\hat{}((lbl = RefCell)^{\downarrow}))$
  $((insertSheet\ clas\ (lbl, [\,])) \rhd pullUpAllRefs)\ tx$
```

Recall that our references are always local, that is, they can only exist within the type they are associated with. So, it is not possible to insert a column that references a part of the existing spreadsheet. To overcome this, we first create the reference with undefined functions and auxiliary type ($\perp$) and then we set these values to the intended ones.

$$\begin{array}{l} \mathit{setFormula} :: \mathit{Type}\ b \to \mathit{PF}\ (a \to \mathit{RefCell}) \to \mathit{PF}\ (a \to b) \to \mathit{Rule} \\ \mathit{setFormula}\ \mathit{tb}\ \mathit{fRef}\ \mathit{tRef}\ (\mathit{Ref}\ \_\ \_\ \_\ t) = \mathit{return}\ (\mathit{View}\ \mathit{idrep}\ (\mathit{Ref}\ \mathit{tb}\ \mathit{fRef}\ \mathit{tRef}\ t)) \end{array}$$

This rule receives the auxiliary type, *tb*, the two functions representing the reference projections, *fRef* and *tRef*, and adds them to the type existing type *t*. A complete rule to insert a column with a reference is defined as follows:

$$\begin{array}{l} \mathit{insertFormula} = \\ \quad (\mathit{once}\ (\mathit{insertCol}\ \texttt{"After Tax"}\ \mathit{FRef})) \triangleright (\mathit{setFormula}\ \mathit{auxType}\ \mathit{fromRef}\ \mathit{toRef}) \end{array}$$

Following the original idea described in Section 6.2, we want to introduce a new column with the tax tariff. In this case, we want to insert a column in an existing block and thus our previous rule will not work. For these cases we write a new rule:

$$\begin{array}{l} \mathit{insertColIn} :: \mathit{String} \to \mathit{Formula1} \to \mathit{Rule} \\ \mathit{insertColIn}\ \mathit{lbl}\ (\mathit{FValue}\ v)\ \mathit{tx} \mid \mathit{isBlock}\ \mathit{tx} = \mathbf{do} \\ \quad \mathbf{let}\ \mathit{block} = \mathit{lbl}\hat{\ }(\mathit{lbl} = v) \\ \quad ((\mathit{insertBlock}\ \mathit{block}\ (\mathit{lbl}, v)) \triangleright \mathit{pullUpAllRefs})\ \mathit{tx} \end{array}$$

This rule is similar to the previous one but it creates a block (not a class) and inserts it also after a block. The reasoning is analogous to the one in *insertCol*.

To add the two columns `"Tax tariff"` and `"After tax"` we can use the rule *insertColIn*, but applying it directly to our running example will fail since it expects a block and we have a spreadsheet. We can use the combinator *once* to achieve the desired result. This combinator tries to apply a given rule somewhere in a type, stopping after it succeeds once. Although this combinator already existed in the 2LT framework, we extended it to work for spreadsheet models. Assuming that the column `"Tax tariff"` was already inserted, we can run the following functions:

$$\begin{array}{l} *\mathit{ghci}{>}\ \mathbf{let}\ \mathit{formula} = \mathit{FFormula}\ \times\ [\mathit{FRef}, \mathit{FRef}] \\ *\mathit{ghci}{>}\ \mathit{once}\ (\mathit{after}\ \texttt{"Tax tarif"}\ (\mathit{once}\ (\mathit{insertColIn}\ \texttt{"After Tax"}\ \mathit{formula}))) \\ \qquad\qquad \mathit{budget} \\ \ldots \\ (\texttt{"Cost"} \mid \texttt{"Tax tariff"} \mid \texttt{"After tax"}\hat{\ }(\texttt{"after tax"} = \mathit{formula}) \mid \texttt{"Total"}) \\ \hat{\ }(\texttt{"cost"} = 0 \mid \texttt{"tax tarif"} = 0 \mid \texttt{"total"} = \mathit{totalFormula}) \\ \ldots \end{array}$$

Note that, above result is not quite right. The block inserted is a vertical composition and is inserted in a horizontal composition. The correct would be to have its top and bottom part on the top and bottom part of the result, as defined below:

$(\texttt{"Cost"} \shortmid \texttt{"Tax tariff"} \shortmid \texttt{"After tax"} \shortmid \texttt{"Total"})\hat{}$
$(\texttt{"cost"} = 0 \shortmid \texttt{"tax tarif"} = 0 \shortmid \texttt{"after tax"} = \mathit{formula} \shortmid \texttt{"total"} =$
$\quad \mathit{totalFormula})$

To correct these cases, we designed a layout rule, *normalize*, explained in Section 6.4.3.

Make it Expandable It is possible to make a block in a class expandable. For this, we created the rule *expandBlock*. Its formal definition is shown next:

$$\begin{tikzcd}(label : clas) \arrow[r, bend left, "id \times tolist"] \arrow[r, phantom, "\leqslant"] & (label : (clas)^{\downarrow}) \arrow[l, bend left, "id \times head"]\end{tikzcd}$$

Its implementation is as follows:

```
$expandBlock :: String \rightarrow Rule$
$expandBlock\ str\ (label : clas) \mid compLabel\ label\ str = \textbf{do}$
  $\textbf{let}\ rep = Rep\ \{to = id \times tolist, from = id \times head\}$
  $return\ (View\ rep\ (label : (clas)^{\downarrow}))$
```

It receives the label of the class to make expandable and updates the class to allow repetition. The result type constructor is $\cdot : (\cdot)^{\downarrow}$; the *to* function wraps the existing block into a list, *tolist*; and the *from* function takes the head of it, *head*. We developed a similar rule to make a class expandable. This corresponds to promote a class c to $c^{\rightarrow}$. We do not show its implementation here since it is quite similar to this one.

Split It is quite common to move a column in a spreadsheet from one place to another. The rule *split* copies a column to another place and substitutes the original column values by references to the new column (similar to create a pointer). The rule to move part of the spreadsheet is presented in Section 6.4.3. The first step of *split* is to get the column that we want to copy:

```
$getColumn :: String \rightarrow Rule$
$getColumn\ h\ t\ (l'\hat{}b1) \mid h \equiv l' = return\ (View\ idrep\ t)$
```

If the corresponding label is found, the vertical composition is returned. Note that, as in other rules, this rule is intended to be applied using the combinator *once*. As we said, we aim to write local rules that can be used at any level using the developed combinators.

In a second step, the rule creates a new a class containing the retrieved block:

$$\textbf{do}\ \mathit{View}\ s\ c' \leftarrow \mathit{getBlock}\ \mathit{str}\ c$$
$$\quad \textbf{let}\ \mathit{nsh} = \mid \mathit{str} : (c')^{\downarrow}$$

The last step is to transform the original column that was copied into references to the new column. The rule $\mathit{makeReferences} :: \mathit{String} \rightarrow \mathit{Rule}$ receives the label of the column that was copied (the same as the new column) and creates the references. We do not shown the rest of the implementation because it is quite complex and it is not essential to understand our techniques.

Let us consider the following part of our example:

```
budget =
  ... ("Cost" | "Tax tariff" | "After tax" | "Total")^
    ("cost" = 0 | "tax tarif" = 0 | "after tax" = formula | "total" =
      totalFormula) ...
```

If we apply the *split* rule (with the help of *once*) to it we get the following new model:

```
* ghci> once (split "Tax tariff") budget
...
("Cost" | "Tax tariff" | "After tax" | "Total")^
("cost" = 0 | "tax tarif" = 0 | RefCell | "total" = totalFormula)
|
(| "Tax tariff" : (("Tax tariff"^"tax tarif" = 0))↓)
```

Split Functional Dependencies As we saw in previous chapters, it is quite common to have columns with repeated data and inducing functional dependencies (see, for example, Figure 5.2). This duplication of data can be the source of errors and inconsistencies in the spreadsheet. A possible solution is to refactor the spreadsheet creating a new table with the repeated information cleaned and substitute the existing columns by references to the new table. We present a rule that automates this process, $\mathit{splitFD} :: [\mathit{String}] \rightarrow [\mathit{String}] \rightarrow \mathit{Rule}$.

Given two lists of column names, the antecedent and the consequent of the functional dependency, this rule creates a new class with the given columns and composes the existing sheet with the new class. At the value-level, the repeated data in the new table is removed. The old consequent columns are all removed, and the antecedent ones are updated to reference the new table. This is possible because we know that there is a functional dependency imposing a relationship between antecedent and consequent columns.

Although this is a complex operation, this will facilitate the maintenance of the spreadsheet. In fact we still can improve this result. Remember that, in a database setting the relationship that we create when we update the antecedent columns to references to the new table is called a *foreign key*. Since 2LT has the expressibility of the invariants, we can add one invariant to this spreadsheet imposing that the columns updated to references just have references to the new column where the antecedent is, that is, a foreign key.

Moreover, given the flexibility of the invariants, we can add any kind of constraints to spreadsheet models, for example primary keys. In fact, the *splitFD* rule ensures that the new table created has as primary key the antecedent columns.

6.4.3 Layout Rules

In this section we describe rules focused on the layout of spreadsheets, that is, rules that do not add/remove information to/from the model.

Change Orientation The rule *toVertical* changes the orientation of a block from horizontal to vertical.

$$\begin{array}{l} \mathit{toVertical} :: \mathit{Rule} \\ \mathit{toVertical}\ (a \mid b) = \mathit{return}\ (\mathit{View}\ \mathit{idrep}\ (a\hat{}\, b)) \end{array}$$

Note that, since our value-level representation of these compositions are pairs, the *to* and the *from* functions are simply the identity function. The needed information is kept in the type-level with the different constructors. A rule to do the inverse was also designed but since it is quite similar to this one, we do not show it here.

Normalize Blocks When applying some transformations, the resulting types may not have the correct shape. A common example is to have as result the following type:

$A \mid B\,\hat{}\,C \mid D\,\hat{}$
$E \mid F$

Most of the times, the correct result is the following:

$A \mid B \mid D\,\hat{}$
$E \mid C \mid F$

The rule *normalize* tries to match these cases and correct them. The types are the ones presented above and the witness functions are combinations of π_1 and π_2.

```
$normalize :: Rule$
$normalize\ (a \mid b\,\hat{}\,c \mid d\,\hat{}\,e \mid f) =$
  **let** $tof = id \times \pi_1 \times id \circ \pi_1 \triangle \pi_1 \circ \pi_2 \triangle \pi_2 \circ \pi_1 \circ \pi_2 \times \pi_2$
      $fromf = \pi_1 \circ \pi_1 \triangle \pi_1 \circ \pi_2 \times \pi_1 \circ \pi_2 \triangle \pi_2 \circ \pi_2 \circ \pi_1 \triangle id \times \pi_2 \circ \pi_2$
  $return\ (View\ (Rep\ \{to = tof, from = fromf\})\ (a \mid b \mid d\,\hat{}\,e \mid c \mid f))$
```

Although the migration functions seem complex, they just rearrange the order of the pair so they have the correct order.

Shift It is quite common to move parts of the spreadsheet across it. We designed a rule to shift parts of the spreadsheet in the four possible directions. We show here part of the *shitRight* rule, which, as suggested by its name, shifts part of the spreadsheet to the right. In this case, a block is moved and an empty block is left in its place.

```
$shitRight :: Type\ a \rightarrow Rule$
$shitRight\ ta\ b1 \mid isBlock\ b1 =$ **do**
  $Eq \leftarrow teq\ ta\ b1$
  **let** $rep = Rep\ \{to = pnt\ (\bot :: EmptyBlock) \triangle id, from = \pi_2\}$
  $return\ (View\ rep\ (EmptyBlock \mid b1))$
```

The function *teq* verifies if two types are equal. This rule receives a type and a block, but we can easily write a wrapper function to receive a label in the same style of *insertCol*.

Another interesting case of this rules occurs when the user tries to move a block (or a sheet) that has a reference.

```
$shitRight\ ta\ (Ref\ tb\ frRef\ toRef\ b1) \mid isBlock\ b1 =$ **do**
  $Eq \leftarrow teq\ ta\ b1$
```

```
    let rep = Rep {to = pnt (⊥ :: EmptyBlock)△id, from = π₂}
    return (View rep (Ref tb (frRef ∘ π₂) (toRef ∘ π₂) (EmptyBlock ı b1)))
```

As we can see in the above code, the existing reference projections must be composed with the selector π_2 to allow to retrieve the existing block *b1*. Only after this, it is possible to apply the defined selection reference functions.

Move Blocks A more complex task is to move a part of the spreadsheet to another place. We present next a rule to move a block.

```
moveBlock :: String → Rule
moveBlock str c = do View s c′ ← getBlock str c
                     let nsh = | str : c′
                     View r sh ← once (removeRedundant str) (c ı nsh)
                     return (View (comprep s r) sh)
```

After getting the intended block and creating a new class with it, we need to remove the old block using *removeRedundant*.

```
removeRedundant :: String → Rule
removeRedundant s (s′) | s ≡ s′ = return (View rep EmptyBlock)
  where rep = Rep {to = pnt (⊥ :: EmptyBlock), from = pnt s′}
```

This rule will remove the block with the given label leaving an empty block in its place.

6.5 Conclusions

In this chapter we have presented an approach for disciplined model-driven evolution of spreadsheets. The approach takes as starting point the observation that spreadsheets can be seen as instances of a *ClassSheet* models capturing the business logic of the spreadsheet. We have extended the calculus for coupled transformations of the 2LT platform to this spreadsheet model. An important novel aspect of this extension is the treatment of references. In particular, we have made the following contributions:

- We have provided a model of spreadsheets in the form of a GADT with embedded point-free function representations. This model is reminiscent of the *ClassSheet*;

- We have shown how to represent references and transform references in a type-safe way. This result can lead to future applications of 2LT variants that target, for example, object-oriented languages;
- We have defined a coupled transformation system in which transformations at the level of spreadsheet models are coupled with corresponding transformations at the level of spreadsheet data/instances. This system combines strategy combinators known from strategic programming with spreadsheet-specific transformation rules;
- We have illustrated our approach with a number of specific spreadsheet refactorings to perform the evolution of spreadsheets.

The rules here presented are implemented in the HAEXCEL framework consisting of a set of libraries providing functionality to load (from different formats), transform, infer spreadsheet models (for example, *ClassSheet*), and, now, perform the co-evolution of such models their (spreadsheet) instances.

Chapter 7

End-user Validation of Model-based Spreadsheets

Summary

Spreadsheets are widely used by end users, and studies have shown that most end-user spreadsheets contain non-trivial errors. To improve end user's productivity, recent research proposes the use of a model-driven engineering approach to spreadsheets.

In this chapter we conduct a systematic empirical study to assess the effectiveness and efficiency of this approach. A number of end users worked with two different model-based spreadsheets, and we present and analyze here the results achieved.

7.1 Introduction

Spreadsheets are prone to errors: numerous studies have shown that the high rate of production of spreadsheets is accompanied by an alarming high rate of errors (Panko 2000; Powell and Baker 2003; EuSpRIG 2011). Some studies report that up to 90% of real-world spreadsheets contain errors (Rajalingham *et al.* 2001).

The software engineering community has recently done some work to make spreadsheets safer, less error-prone, and to improve end user's productivity (Engels and Erwig 2005; Erwig *et al.* 2005; Abraham and Erwig 2006a; Cunha *et al.* 2009a,b, 2010). One of the promising solutions advocates the use of a *Model-Driven Engineering* (MDE) approach to spreadsheets. In such an approach, a business model of the spreadsheet data is defined, and then end users are guided to introduce data that conforms to the defined model. Indeed, several models to represent the business logic of the spreadsheet have been proposed, namely, templates (Erwig *et al.* 2005; Abraham and Erwig 2006a), *ClassSheets* (Engels and Erwig 2005), or relational models as we propose in Chapter 3. In fact, we proposed in Chapter 3 several techniques to infer such models from a (legacy) spreadsheet.

Although all these works claim that by using an MDE approach end-users productivity is improved, the reality is that there is no detailed evaluation study that supports this claim. In this chapter, we present an empirical study that we have conducted with the aim of analyzing the practical influence of using models in end-users spreadsheet productivity. In this study we consider two different model-based spreadsheets: *refactored* spreadsheets, as we propose in Chapter 5 and spreadsheets with *edit assistance*, as we propose in Chapter 4. From now on, the former will be referred as *refactored* and the latter as *visual*, as apposed to the original, more usual model, referred as *original*. We assess the productivity of end users when introducing, updating and querying data in those two model-based spreadsheets and in a traditional one.

In this chapter we wish to answer the following research questions:

RQ1 *Do end users introduce fewer errors when they use one of the model-based spreadsheet versus the original unmodified spreadsheet?*

RQ2 *Are end users more efficient using the model-based spreadsheets?*

RQ3 *Do particular models support particular tasks better, leading to fewer errors in those tasks?*

The study we conducted to answer these questions is necessary and useful, since it is based on a sound experimental setting, and thus, allow us to draw sound conclusions for further studies on how to improve spreadsheet end user's productivity.

This chapter is organized as follows. In Section 7.2 we describe the design of our study. We present and analyze in detail the results of our study in Section 7.3. Several threats to validity are discussed in Section 7.4. Finally, we draw our conclusions in Section 7.5.

7.2 Study Design

As suggested in (Perry *et al.* 2000) we organized the study as follows:

1. *Formulating hypothesis to test*: we spent a considerable amount of time organizing our ideas and finally formulating the hypothesis presented in this work: we hypothesize that model-based spreadsheets can help end users to commit fewer errors when editing and querying spreadsheets.

2. *Observing a situation*: next, we gathered participants willing to be involved in the study and we ran the study itself, once we got enough and appropriate qualified participants. During the study, we screen casted the participants' computers and afterwards we collected the spreadsheets they worked on.

3. *Abstracting observations into data*: we computed a series of statistics, that we present in Section 7.3, over the spreadsheets participants developed during the study: we graded their performance and measured the time they took to perform the proposed tasks. All the data we used is available at the *SpreadSheets as a Programming Paradigm* project web page `http://ssaapp.di.uminho.pt`. The tasks and the spreadsheets participants received are also available in that page.

4. *Analyzing the data*: the enormous collection of data that we gathered was later systematically analyzed. This analysis is also presented in this chapter, in Section 7.3.

5. *Drawing conclusions with respect to the tested hypothesis*: based on the results obtained, we finally drawn some conclusions. We were also able to suggest some future research paths based on our work, which are presented in Section 7.5.

Our study aimed to answer if participants were able to perform their tasks with more accuracy and/or faster given the experimental environments. We used a within subjects design, meaning that each participant received a task list for each of three spreadsheet environments. Participants were asked to do various tasks in each spreadsheet like, for example, data entry, modifications to existing data, and calculations of the data in the spreadsheet. They were encouraged to work as quickly as possible, but were not given time limits for any specific spreadsheet.

7.2.1 Methodology

Participants started the study by filling out a background questionnaire so we could collect their area of study and previous experience with spreadsheets, other programming languages and English comfort (because Portuguese was the mother language for all participants). An introduction to the study was given orally in English, this was explicitly not a tutorial for the different environments because the goal was to see if even without any introduction to the various experimental spreadsheets the participants would still be able to understand and complete the spreadsheet tasks. The participants were asked to work as quickly and accurately as possible. Since the order of the spreadsheets was randomized, they were told that the others sitting around them might appear to be moving faster, but that some tasks were shorter than others. We randomized the order of the spreadsheets in the which the different models were presented to counter the learning effect. After 2 hours participants were stopped if they were not already finished. Following the tasks they had a post session questionnaire which contained questions assessing their understanding of the different spreadsheet environments, (3 questions for *refactored* and 4 for *visual*), plus their own preference. We also asked how confident they were that they had correctly completed the tasks in the three task lists. Correct answers could only be given by participants having understood the running models. Grading the questionnaires was done as follows: a correct answer receives total points; an incorrect answer receives zero points and an answer that is not incorrect nor (totally) correct receives half of the points. We recorded the users screens using screen capture technology. At the end of the study the users completed spreadsheets were saved and graded for later analysis.

7.2.2 Participants

Recruitment was conducted through a general email message to the university, asking for students with spreadsheet experience and comfort with English. Of the hundreds that responded (there was a compensation involved), participants were selected based on spreadsheet experience, comfort with English, and majors outside of computer science and engineering. In total, 38 participants finished the study with data we were able to use (25 females, 11 males, and 2 who did not answer about their gender). Two participants did not try to solve one of the proposed tasks; for these participants, we included in the study only the tasks they undertook. A few participants' machines crashed and therefore they were eliminated from the study. The majority of participants were between 20-29 years of age, with the remaining under 20. All were students at the university. About 2/3 were working on their Bachelor's degree and the remaining on their Masters. None were studying computer science or engineering and the most represented majors were medicine, economics, nursing and biology. A variety that is good for representing the end-user population of spreadsheet users. Table 7.1 summarizes our participants.

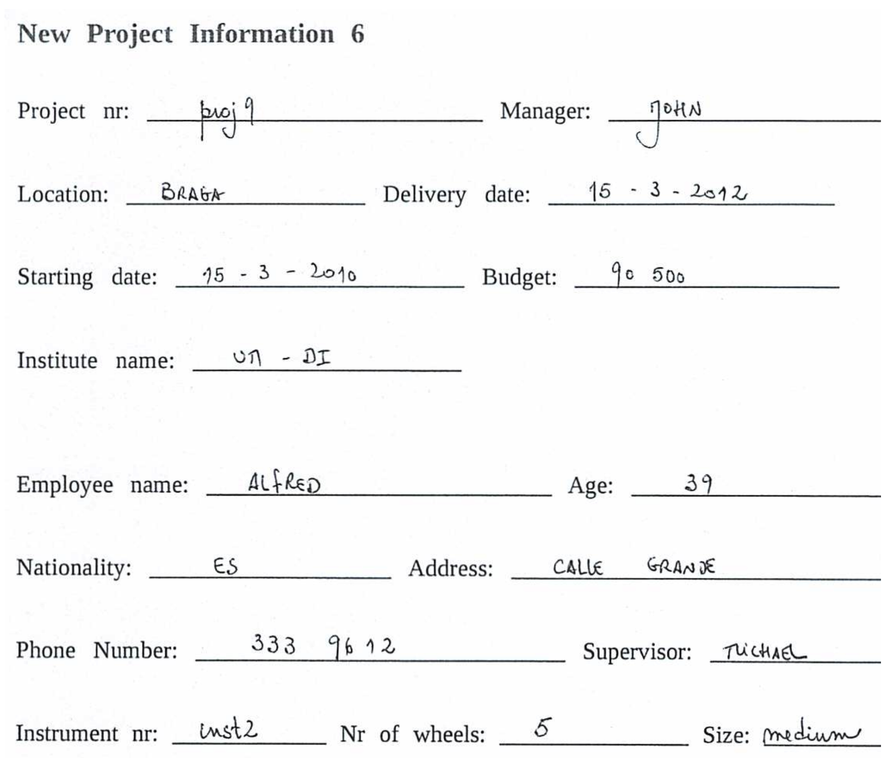
What	How many
Participants	38
Female	25
Male	11
Not answer	2
In baccalaureate	25
In masters	13

Table 7.1: Summary of the participants' data.

7.2.3 Tasks

The tasks were designed to highlight both the areas which the spreadsheet environments may be an advantage in, and areas where spreadsheets are known to be problematic. The tasks were 1) add new information to the spreadsheet, 2) edit existing data in the spreadsheets and 3) do some calculations using the data in the spreadsheets. Figure 7.1 illustrates an example of a sheet that participants received containing data for inserting.

Some of the tasks asked the users to add many new rows of data, with the aim of a repetitive task being common in real-world situations. As we were designing the tasks, we imagined a type of data entry office scenario, where an office worker might receive on paper data which was initially filled out on a paper form and needed to be entered into a spreadsheet. This first task of data entry, in theory, should be fastest (and done with fewest entry errors) in the *refactored* spreadsheet environment. The second task, of making changes to existing data within a spreadsheet should also be easier within a *refactored* spreadsheet environment, since the change only needs to be made in one location, and therefore there would be less chance of forgetting to change



New Project Information 6

Project nr: proj 9 Manager: JOHN

Location: BRAGA Delivery date: 15 - 3 - 2012

Starting date: 15 - 3 - 2010 Budget: 90 500

Institute name: UM - DI

Employee name: ALFRED Age: 39

Nationality: ES Address: CALLE GRANDE

Phone Number: 333 96 12 Supervisor: MICHAEL

Instrument nr: inst2 Nr of wheels: 5 Size: medium

Figure 7.1: Example of a sheet that participants received containing data for inserting.

it. The final task was to do some calculations using the data in the spreadsheet, such as averages, etc. This task was added because of the frequency of problems with formulas in spreadsheets.

One of the spreadsheets used in the study, PROPERTIES, stores information about a properties renting system and was adapted from (Connolly and Begg 2001) (also used in Chapter 5). This spreadsheet has information about renters, properties and their owners as well as the dates and prices of the rents.

A second spreadsheet, DISHES, contains information about sells of detergents to dishwashers. Information about the detergents, prices and the stores where they are sold is present on this spreadsheets. This spreadsheet was adapted from (Powell and Baker 2003) (also used in Chapter 3).

The last spreadsheet, PROJECTS, stores information about projects, like the manager and delivery date, employees assigned to them and the instruments used. This spreadsheet was adapted from (Alhajj 2003) (also used in Chapter 2).

In the task list for DISHES, 82% of the tasks consist of inserting new data, 3% are editing tasks and 15% involve calculations over the data in the spreadsheet. In the task list for PROJECTS, 98% of the tasks are for inserting new data, 1% for edition and 1% for calculations. Finally, for PROPERTIES, inserting data tasks are 67% of the total, whereas data editing and calculation tasks are 2% and 31% of the total, respectively.

Grading the participants' performance was done as follows. For tasks involving adding new data to the spreadsheet or performing calculations over spreadsheet data, whenever a participant executes a task as we asked him/her to, he/she is awarded 100% of the total score for that task; on the contrary, if the participant does not at all try to solve a particular task, he/she gets no credit for that. An intermediate situation occurs when participants try to solve a task, but fail to successfully conclude it in its entirety. In this case, the participant is awarded 50% of the score for that task. For tasks involving editing spreadsheet data, a value in the interval 0% – 100% is awarded according to the participants' success rate in such tasks (that we can accurately measure).

Table 7.2 presents the number of participants that worked on each spreadsheet and each model.

	original	*refactored*	*visual*	Total
DISHES	12	13	12	37
PROJECTS	11	13	13	37
PROPERTIES	14	11	13	38
Total	37	37	38	

Table 7.2: Number of participants that worked on each spreadsheet/model.

As expected, the distribution of the models and spreadsheet applications by the participants is quite homogeneous.

7.3 Analyzing End-users Performance

We divide the presentation of our empiric results under two main axes: effectiveness and efficiency. In studying effectiveness we want to compare the three running models for the percentage of correct tasks that participants produced in each one. In studying efficiency we wish to compare the time that participants took to execute their assigned tasks in each of the different models. We start by effectiveness.

7.3.1 Effectiveness

Each participant was handed three different lists of tasks (inserting, editing and querying data) to perform on three different spreadsheets (DISHES, PROJECTS and PROPERTIES). Each spreadsheet, for the same participant, was constructed under a different model (*original*, *refactored* or *visual*).

For each spreadsheet, and for each model, we started by analyzing the average of the scores obtained by participants. We shown in Figure 7.2 those results.

	original	*refactored*	*visual*
DISHES	86%	76%	78%
PROJECTS	73%	68%	78%
PROPERTIES	75%	64%	62%

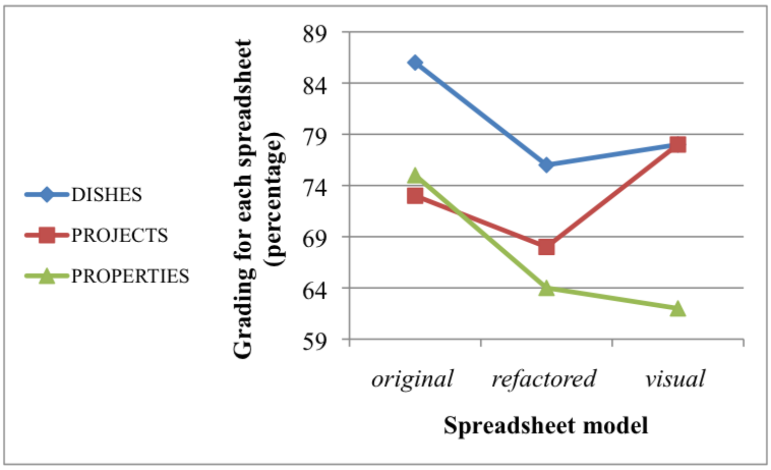


Figure 7.2: Global effectiveness results.

We notice that no spreadsheet model is the best for all spreadsheets in terms of effectiveness. Indeed, we may even notice that spreadsheets in the traditional style, the *original* model, turned out to be the best for both the DISHES and PROPERTIES spreadsheets. The *visual* model suited the best for the PROJECTS spreadsheet.

In the same line of reasoning, there is no worse model: *refactored* spreadsheets achieved the worst results for the DISHES and PROJECTS spreadsheets and the *visual* model got the lowest average scores for PROPERTIES. Nevertheless, these results seem to indicate that the models that we have developed in Chapter 4 and in Chapter 5 are not effective in reducing the number of errors in spreadsheets, since one of them is always the model getting the lowest scores. This first intuition, however, deserves further development.

For once, on the theoretical side, one may argue that the *original* spreadsheet model is, without a doubt, the *model* that end users are accustomed to. Recall that in the study, we opted to leave out participants with computer science backgrounds, who could be more sensible to the more complex models *refactored* and *visual*, preferring to investigate such models on traditional users of spreadsheets. On the other hand, we remark that these more complex models where given no introductory explanation; a part of our study was also to learn whether or not they could live on their own.

Our next step was then to investigate whether the (apparent) poor results obtained by complex models are due to their own nature or if they result from participants not having understood them. In order to fully realize this, we studied the participations that did not achieve a score of at least 50%, which are distributed by spreadsheet model as shown in Table 7.3:

original	0%
refactored	25%
visual	21%

Table 7.3: Participations graded under 50%.

While in the *original* model no participation was graded under 50%, we can see that this was not the case for *refactored* and *visual*, which may have degraded their overall average results. For these participations, we then analyzed the questionnaire that they were asked to fill in after the session. In Table 7.4, we present the average classifications, in percentage, for the post session questionnaires, for participations in the study that were graded under 50%.

refactored	24%
visual	31%

Table 7.4: Grading of post-session questionnaires for participations graded under 50%.

These results show that participants obtaining poor gradings on their effectiveness also got extremely poor gradings for their answers to the questions assessing how they understood the models. Indeed, we can see that such participants were not, in average, able to answer correctly to (at least) two thirds of the questions raised in the post session questionnaire. From such results we can read that 1/4 of participants was not able to understand the more complex models, which might have caused a degradation of the global effectiveness results for these models. This also suggests that, if these models are to be used by an organization, it is necessary to take some time to introduce them in order to achieve maximum effectiveness. Nevertheless, even without giving this introduction in our study, the results show that the models are competitive in terms of effectiveness: at most, they are 13% worst than the *original* model, and for one of the spreadsheets, the *visual* model even got the best global effectiveness results.

Effectiveness by Task Type

Next, we wanted to realize how effective the models are to perform each of the different types of tasks that we have proposed to participants: adding new data to a spreadsheet, or *data insertion*, changing the data on a spreadsheet, or *data editing* and performing some calculations over the spreadsheet data, or *statistics*.

i) *Data insertion:* The results presented in Figure 7.3 show, for each model, how effective participants were in adding new information to the spreadsheets.

	original	*refactored*	*visual*
DISHES	91%	90%	81%
PROJECTS	76%	60%	75%
PROPERTIES	86%	67%	68%

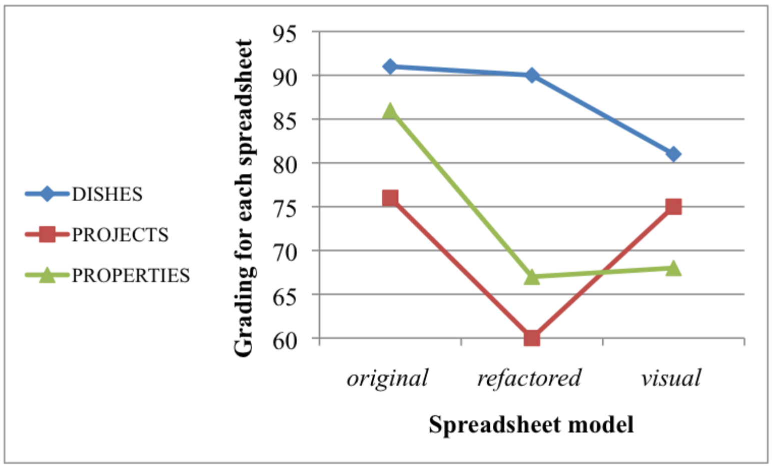


Figure 7.3: Effectiveness results for data insertion.

The *original* model turned out to be the most effective, for all three spreadsheets, being closely followed by *refactored* and *visual* for DISHES, and by *visual* for PROJECTS. The *refactored* model, for PROJECTS, and the models *refactored* and *visual*, for PROPERTIES, proved not to be competitive for data insertion, in the context of the study. Again, we believe that this in part due to these models not having been introduced previously to the study: the insertion of new data is the task that is most likely to benefit from totally understanding of the running model, and also the one that can be otherwise most affected. This is confirmed by the effectiveness results observed for other task types, that we present next.

ii) *Data editing:* Now, we analyze the effectiveness of the models for editing spreadsheet data. The results presented in Figure 7.4 show that once a spreadsheet is populated, we can effectively use the models to edit its data.

	original	*refactored*	*visual*
DISHES	91%	82%	82%
PROJECTS	54%	62%	50%
PROPERTIES	65%	98%	48%

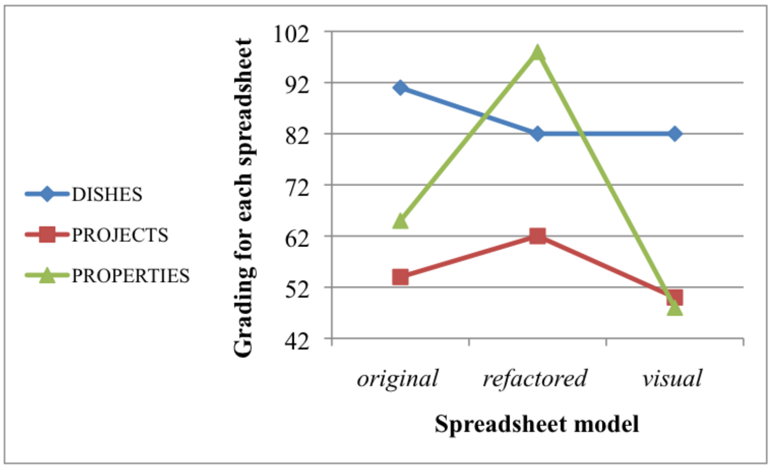


Figure 7.4: Effectiveness results for data editing.

This is the case of the *refactored* model for PROJECTS and specially for PROPERTIES. The *original* model is the most effective in data editing for DISHES. The *visual* model is comparable to *refactored* for DISHES, but for all other spreadsheets, it always achieves the lowest scores among the three models.

iii) *Statistics:* Finally, we have measured the effectiveness of the models for performing calculations over spreadsheet data, obtaining the results shown in Figure 7.5.

	original	*refactored*	*visual*
DISHES	52%	37%	57%
PROJECTS	19%	76%	13%
PROPERTIES	44%	57%	51%

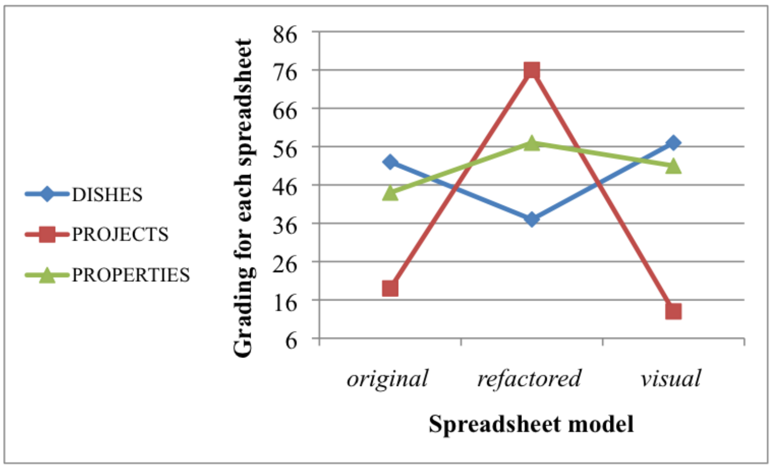


Figure 7.5: Effectiveness results for statistical calculations.

We can see that *visual* obtained the best results for DISHES and that *refactored* obtained the best results for both PROJECTS and PROPERTIES. We can also see that, all models obtained the worst results for exactly one spreadsheet.

Results from *i*), *ii*) and *iii*) confirm that the models are competitive against the *original* model. On the other hand, these results allow us to draw some new conclusions: if the models are going to be used within an organization, it may not always be necessary to introduce them prior to their use. Indeed, if an organization mostly edits spreadsheet data or computes new values from such data, and does not insert new data, then the models, and specially *refactored*, may deliver good results even when they are not explicitly explained (as it was the case in our study). These results also show that it is in the data insertion tasks that the models need to be better understood by end users in order to increase effectiveness.

7.3.2 Efficiency

In this section, we analyze the efficiency results obtained in our study by the models that we have been considering in this chapter.

We started by measuring, for each participant, and for each spreadsheet, the time elapsed from the moment participants started reading the list of tasks to undertake until the moment they completed the tasks proposed for that particular spreadsheet and moved on to a different spreadsheet or concluded the study. We are able to calculate these times by looking at the individual screen activity that was recorded during the study, for each participant: the participant stopping interacting with the computer signals the end of his/her work on a spreadsheet. The measured period therefore includes the time that participants took trying to understand the models they received each spreadsheet in. Figure 7.6 presents the average of the overall times, for each spreadsheet and for each model.

	original	*refactored*	*visual*
DISHES	35′	32′	28′
PROJECTS	39′	40′	41′
PROPERTIES	37′	36′	40′

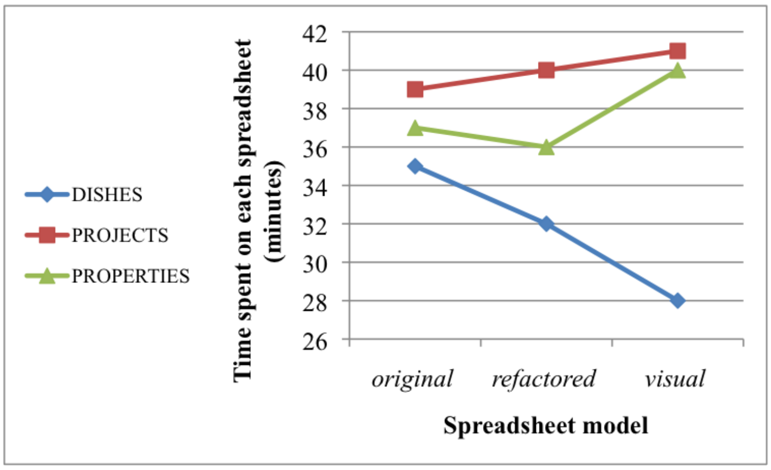


Figure 7.6: Global efficiency results.

We can see that the models *refactored* and *visual* are competitive in terms of efficiency against the *original* model. Indeed, participants performed fastest for the DISHES spreadsheets in the *visual* model, and fastest, by a marginal factor, for the

PROPERTIES spreadsheet in the *refactored* model. The *original* model got the best efficiency measurements for the PROJECTS spreadsheets, also by a marginal factor. Again, note that no introduction to *refactored* or *visual* preceded the study. Therefore, it is reasonable to assume that, for these models, the results in Figure 7.6 include some time overhead. In an attempt to measure this overhead, which is a consequence of participants having to analyze a new model, we extracted some more information out of the results of our study (particularly from the participants' individual screen activity record). Indeed, we measured the time elapsed from the moment participants started reading, for each spreadsheet, the list of tasks to perform, until the moment they actually began editing the spreadsheet. We assume that this period corresponds exactly to the overhead of understanding each model (obviously increased by the time spent reading the list of tasks, which we are not able to isolate further, but that should be constant for any spreadsheet model, since the task list does not change with the model). The average results obtained are presented in Table 7.5.

	original	*refactored*	*visual*
DISHES	$2'$	$6'$	$1'$
PROJECTS	$2'$	$4'$	$2'$
PROPERTIES	$2'$	$2'$	$2'$

Table 7.5: Average overhead results.

We notice that there is a constant average overhead of 2 minutes for almost all models and all spreadsheets, with the most significant exceptions occurring for the *refactored* model, for both the DISHES and the PROJECTS spreadsheets. In these cases, we can clearly notice an important time gap, which provides some evidence that *refactored* is most likely the hardest model to understand. This also comes in line with previous indications that the merits of the spreadsheet models can be maximized if we take the time to explain them to end users. For the particular case of efficiency, this means that the results shown in Figure 7.6 could be further improved for the more complex models, and particularly for the *refactored* model.

7.4 Threats to Validity

As suggested by Perry *et al.* (2000), we discuss three types of influences that might limit the validity of our study.

7.4.1 Construct Validity

Do the variables and hypotheses of our study accurately model the research questions?

- *Measuring the time overhead*: When studying efficiency, we measured the overhead of understanding each model as the period of time that participants stopped interacting with a particular spreadsheet and started editing the following one. In this period, it might have been the case that participants, instead of being really focused on understanding the new model, took the time to do something else, like resting, for example. This could affect the conclusions that we draw, in terms of efficiency. However, during the entire study, participants where always supervised by at least two authors, who observed that this was not the case. Even if we were not able to spot a small number of such occurrences, the differences in the results computed should be minimal and therefore they should not affect our conclusions.

- *Original model*: In our study, we have used three spreadsheets that we have assumed to be in the *original* model. What we are saying is that these three spreadsheets are representative of the spreadsheets normally defined by end users. Although this set of spreadsheets may be too large to be represented by (any) three spreadsheets, we have taken DISHES, PROJECTS and PROPERTIES directly, or with small changes, from other works on general purpose.

7.4.2 Internal Validity

Can changes in the dependent variables be safely attributed to changes in the independent variable?

- *Accuracy of the analysis*: Some of the inferences we make in this chapter deserve further analysis. To some extent, we assume that our models could achieve better results if a tutorial has been given to the participants. In fact, we have no proof of this, but the evidences from the study seem to strongly indicate this fact. A new study is required to prove this, though.

- *Accuracy of measurements*: Each task proposed to participants was graded individually according to the participants' performance. For most of the cases, this was done automatically using *OpenOffice.org* BASIC scripts. These scripts and their results were exhaustively tested and checked. The cases for which an automatic grading was not possible were carefully graded by hand. All grades were validated by two authors and were randomly rechecked. Since we have more than 1400 grades, it is virtually impossible to guarantee full grading accuracy. This could affect the results observed for dependent variables (efficiency and effectiveness) without really the independent variables (the models considered) having changed. Nevertheless, if imprecisions exist in the grades, they should be equally distributed by the three models and thus they should not affect the overall results or our conclusions.

  The measurement of times that lead to the results presented earlier was achieved by individually visualizing the screen cast made during the study for each participant. Being a manual task, and a repetitive one, this is subject to imprecisions. Also, not being able to visualize the actual participants' behavior now may lead us to imprecise measurements. We are confident that, even if the observed results are in fact subject to imprecisions, such imprecisions should be distributed evenly by all measurements and thus do not influence the efficiency results or the conclusions that we draw based on them.

7.4.3 External Validity

Can the study results be generalized to settings outside the study?

- *Generalization*: In this study we used three different spreadsheets from different domains. We believe that the results can be generalized to other spreadsheets, although probably not to all. The models we developed are not restricted to any particular spreadsheet, and thus, the results should be the same if the study was run with a different set of spreadsheets.

- *Industrial usage*: Participants in our study were students who were asked to simulate industrial activity: they received some data on paper that they had to register in a spreadsheet, and to further manipulate. Although we have tried to create conditions similar as possible to reality, it is likely that people could act differently in an industrial/real environment. Nevertheless, we believe that no particular spreadsheet or model should be affected by this. Indeed, if this would really be the case, then it probably would affect all spreadsheets and all models in the same way and thus the overall results apply. We believe that if the study was conducted on an industrial environment, the conclusions should be similar.

7.5 Conclusions

In this chapter, we have presented the results of an empirical study that we conducted in order to assess the practical interest of models for spreadsheets.

According to (Perry *et al.* 2000), three topics deserve further analysis, namely, *accuracy of interpretation*, *relevance* of our study and its *impact*.

- *Accuracy of interpretation*: This study was prepared carefully and a significantly large number of end users participated in it. Our goal here was to guarantee that the results are not unknowingly influenced. For this, it also contributes the fact that we make all the elements of this study available, both in this chapter and online.
- *Relevance*: MDE is one of the most significant research areas in software engineering. We adapted some techniques from this field to spreadsheets and showed that they can bring benefit to end users, and possibly for professional users too.
- *Impact*: Our first results show that MDE can bring benefits for spreadsheet end users. This is a promising research direction, that we believe can be further explored, particularly in contexts similar to the one of this chapter.

From the preparation of the study, from running it and from its results, we can summarize our main contributions as follows:

- We have shown that MDE techniques can be adapted for end-users software;
- We found some empirical evidences that models can bring benefits for spreadsheet end users;

- We proposed a methodology that can be reused in studies similar to the one we have conducted.

Finally, we seek to answer the research questions that we presented in the introduction of this chapter, which correspond exactly to the questions our study was designed to answer.

RQ1 *Do end users introduce fewer errors when they use one of the model-based spreadsheet versus the original unmodified spreadsheet?*

Our observations indicate that model-based spreadsheets can improve end-user effectiveness. Even if this is not always the case, our results also indicate that deeper insight on the spreadsheet models is required to maximize effectiveness. Indeed, we believe that the effectiveness results for *refactored* and *visual* could have been significantly better if these models had been preliminary presented to the participants of our study.

RQ2 *Are end users more efficient using the model-based spreadsheets?*

We observed that, frequently, the more elaborate spreadsheet models allowed users to perform faster. Nevertheless, we were not fully able of isolating the time that participants took trying to understand the models they were working with. So, we believe that the observed efficiency results could also be better for *refactored* and *visual* if they had been previously introduced.

RQ3 *Do particular models support particular tasks better, leading to fewer errors in those tasks?*

Although this was not observed for inserting tasks, the fact is that, for editing and querying data the models did help end users. Furthermore, the results seem to indicate that the inserting data task is the one that benefits the most from better understanding the models.

With this study we have shown that there is potential in MDE techniques for helping spreadsheet end users. The study of these techniques for professional users of spreadsheets seems a promising research topic. Moreover, the use of MDE techniques in other non-professional softwares should also be investigated.

Chapter 8

The HAEXCEL Framework

Summary

In this chapter we present HAEXCEL, *an open source framework we have developed to support the techniques presented in the previous chapters of this thesis. This framework consists of a set of libraries, tools built on top of such libraries and an extension to an open source, widely used spreadsheet system: OpenOffice.org. It allows the users to manipulate, transform and query spreadsheets. We explain in this chapter its architecture and each of its components.*

8.1 Introduction

In the previous chapters we have presented a series of theories and techniques to manipulate spreadsheets. In this section, we present the framework HAEXCEL that we have developed to make all these techniques available for other researchers and end users.

The framework was developed in the HASKELL programming language (Hudak and Fasel 1992; Jones 2003) and its main component is a series of reusable HASKELL libraries capable of manipulating spreadsheets. An overview of the framework can be seen in Figure 8.1.

The arrows in the figure denote HASKELL components that implement various analyses, transformations, and generators. Next, we describe succinctly each component of the framework.

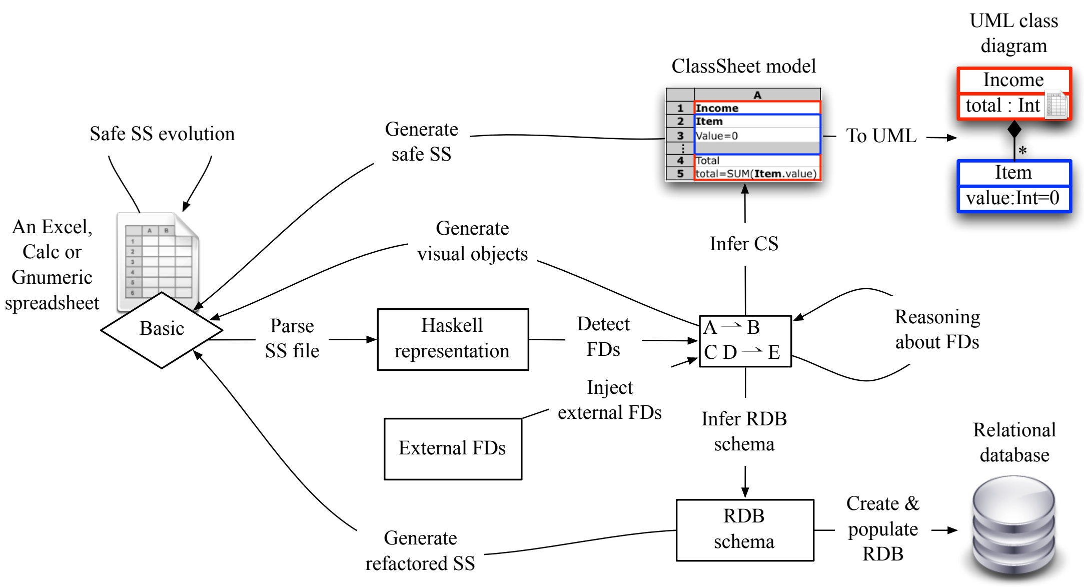


Figure 8.1: Architecture of HAEXCEL.

Manipulating spreadsheets We use the *UMinho Haskell Libraries* (UMHL) to import and export spreadsheets to and from HASKELL. This is represented in Figure 8.1 by the *parse SS (spreadsheet) file*, the *generate safe SS* and the *generate refactored SS* components.

Functional dependencies After importing a spreadsheet, we can use the libraries to infer functional dependencies. A series of functions are available for reasoning about them and to produce a reduced or a normalized set. In Figure 8.1 this is represented by the *detect FDs* and the *inject external FDs* components.

Models From functional dependencies, we can construct a relational model for the spreadsheet. A *ClassSheet* model can also be produced by the framework. From the *ClassSheet*, a UML class diagram can also be obtained. This is represented in the overview picture by the *ClassSheet model*, the *RDB schema* and the *UML class diagram* components.

Edit assistance Using again the libraries and functional dependencies, edit assistance can be generated. An *OpenOffice.org* extension is available so this can be used directly in a spreadsheet. The *generate visual objects* component represents this feature in Figure 8.1.

Refactoring Using a relational schema, the framework can generate a new spreadsheet reflecting the relational constraints. Notice that, the data from the original spreadsheet is included in the resulting spreadsheet. An online tool is also available: given a spreadsheet, it will return a new refactored spreadsheet. From the *RDB schema* to the spreadsheet is represented the *generate refactored SS* component, as illustrated in Figure 8.1.

Migration Using the same relational schema as before, the framework can generate an SQL script that can be imported by any relational database management system. It will create and populate the database with the data from the spreadsheet. An online version is also available: for an input spreadsheet it will return the corresponding SQL script. The *create & populate RDB* component shows this part of the tool in the overview picture.

Evolution We reused the 2LT platform for the implementation of this component. A set of rules is available so spreadsheets can be changed in a safe way. The *safe SS evolution* represents this component in the above figure.

This chapter is organized as follows. In Section 8.2 we explain how to parse spreadsheets into HASKELL and how to export them back to spreadsheet files. In Section 8.3 we explain how to extract and manipulate functional dependencies from the imported spreadsheets. These dependencies are used to infer models as explained in Section 8.4. The generation of edit assistance is presented in Section 8.5. The migration of spreadsheets to databases and the generation of refactored spreadsheets is explained in Section 8.6. Section 8.7 presents the spreadsheet evolution component and Section 8.8 concludes this chapter.

8.2 Manipulating Spreadsheets in HAEXCEL

In order to read spreadsheets into HASKELL, we need to be able to import the formats used by popular spreadsheet systems. Before we discuss how to import/export spreadsheets to HASKELL, let us explain our HASKELL representation for spreadsheets.

8.2.1 Representing Spreadsheets in HASKELL

The representation we present here is based on the *UCheck* project (Abraham and Erwig 2007b). In fact, we reuse part of their HASKELL representation. Columns and rows are represented by integers, starting at 1 (that is, column 1 in HASKELL represents column `A` in a spreadsheet, column 2 represents column `B`, and so on). An address is composed by a column and by a row.

```
type Col = Int
type Row = Int
data Indx = Indx Col Row
```

A value of a cell is represented by the *Fml* data type which assumes several shapes:

```
data Fml = Inp Val                                          -- input (by user)
         | Ref Indx                          -- direct reference to another cell
         | Ref' (String,Row)        -- reference using the label and the row
         | Bin BinOp Fml Fml                 -- application of binary operation
         | Log LogOp Fml Fml                                    -- logic formula
         | Agg BinOp Rng                                -- aggregation formulas
         | AggL LogOp Rng                         -- logic aggregation formulas
         | If Fml Fml Fml                                 -- conditional formula
         | Func String [Fml]  -- general case of formula with name and args.
         | UnkFml                                                -- unknown type
```

Next, wee explain each part of this data type representation:

Input values This type, *Inp Val*, represents the plain values such as integers or strings which are represented by the *Val* type.

```
type Date = (Int,Int,Int)    -- (dd, mm, yyyy)
data Val = I Int                    -- integers
         | F Double          -- floating points
         | S String                  -- strings
         | B Bool                   -- booleans
         | E String         -- error / undefined
         | D Date                      -- dates
         | Unused    -- block out "dead parts"
```

Ref Indx Represents a reference to another cell given by the reference *Indx*.

Ref' (String, Row) Another kind of reference by the column label, represented by the string, and by the row number.

Bin BinOp Fml Fml Binary operation *BinOp* application to two other values. *BinOp* is defined as follows:

data *BinOp* = *Plus* | *Minus* | *Mult* | *Div* | *Avg* | *Exp* -- binary operations

Log LogOp Fml Fml Binary operation *LogOp* application to two other values:

data *LogOp* = *Eq* | *Neq* | *Lt* | *Gt* | *And* | *Or* -- logic operations

Agg BinOp Rng Aggregation formula *BinOp* applied to a range *Rng*. The range is defined as **type** *Rng* = [*Fml*].

AggL LogOp Rng Logic aggregation.

If Fml Fml Fml Conditional operator.

Func String [Fml] Formula termed by the string and with arguments given by the list.

UnkFml Unknown type.

A cell is a pair with an address and a value as explained before. A sheet is a list of cells.

type *Cell a* = (*Indx*, *a*)

newtype *Sheet a* = *Sheet* [*Cell a*]

Usually, we represent a spreadsheet by a list of cell of type *Fml*, that is, by *Sheet Fml*.

8.2.2 Importing Spreadsheets

Having defined a HASKELL data type to store spreadsheets, we need to define functions to read spreadsheets from popular formats. There are two options to import spreadsheets into HASKELL:

***Gnumeric* files** HAEXCEL can read *Gnumeric* (2011) spreadsheet files through the *UMinho Haskell Libraries* using its XML representation. This XML spreadsheet front-end was defined in the context of a graduation final assignment (Miranda 2004). In the context of this thesis, we have integrated the construction of a sheet value into such a front-end. To read this kind of files, the user can use the following function:

$$readSS :: FilePath \rightarrow IO\ [Sheet\ Fml]$$

This function receives a *Gnumeric* file and returns a list with the tables contained in such spreadsheet.

Other formats HAEXCEL can read any kind of spreadsheet files if they are represented as *Comma Separated Values* (CSV) files. The function *csv2ss* has the same signature as *readSS* and thus produces the same result, but receiving a CSV file.

8.2.3 Exporting Spreadsheets

Although we read several formats, we export only to one: *Gnumeric*. The function

$$ss2SSa :: [Sheet\ Fml] \rightarrow FilePath \rightarrow IO\ ()$$

receives a list of sheets, a file to write the spreadsheets and creates a new spreadsheet in the *Gnumeric* format. If the user wants to change the spreadsheet to a different format, the *Gnumeric* can create, both in batch or interactively, different formats such as *Excel* or *OpenOffice.org*.

8.3 Functional Dependencies

In this section, we explain how to use HAEXCEL to infer and manipulate functional dependencies for spreadsheets. First, we introduce the HASKELL representation for schemas and relations.

8.3.1 Extracting Schemas and Relations

The HAEXCEL functions work mainly with schemas and relations, not with spreadsheets as represented before. So, we need to convert the *Sheet* values into the correct ones. To represent attributes, schemas and relations we use the following types:

$\textbf{data}\ \textit{Attribute a} = \textit{Att a}$ -- one attribute

$\textbf{data}\ \textit{Attributes a} = \textit{Atts}\ [\textit{Attribute a}]$ -- a list of attributes

$\textbf{type}\ R\ a = \textit{Attributes a}$ -- a schema

$\textbf{type}\ \textit{Relation a} = [[a]]$ -- a relation: each row/tuple is a list

An attribute (*Attribute*) can be of any type and it is wrapped with the *Att* constructor. A list of attributes (*Attributes*) is composed by elements of type *Attribute* and it is also wrapped with a constructor: *Atts*. A relational schema *R* is a list of attributes. A table/relation (*Relation*) is a list of rows/tuples of the corresponding elements.

The function *sheet2relation* receives a spreadsheet with one table and returns a pair with the relation schema and with the relation:

$\textit{sheet2relation} :: \textit{Sheet Fml} \to (R\ \textit{String}, \textit{Relation String})$

If the spreadsheet consists of more than one table or sheet, we use another function, *sheets2relation*, because references between relations may exist.

$\textit{sheets2relation} :: [\textit{Sheet Fml}] \to [(R\ \textit{String}, \textit{Relation String})]$

As the reader may have noticed, the relations are of type *String*. In fact this is enough for our functions. We calculate the formulas existing in the spreadsheet and replace them by the result values. When formulas are needed, we use the original value of type *Sheet*, which contains the representation of formulas.

8.3.2 Functional Dependencies in HAEXCEL

Functional dependencies are represented in HASKELL as follows:

$\textbf{type}\ \textit{FD a} = (\textit{Attributes a}, \textit{Attributes a})$

$\textbf{type}\ \textit{FDS a} = [\textit{FD a}]$

$\textbf{type}\ \textit{CFD a} = ([\textit{Attributes a}], \textit{Attributes a})$

$\textbf{type}\ \textit{CFDS a} = [\textit{CFD a}]$

A functional dependency *FD* is a pair of attributes, that is, the antecedent and the consequent. A list of functional dependencies is represents by *FDS* and is constructed using the predefined HASKELL lists.

A compound functional dependency *CFD* is a pair with left sets (candidate keys) and with the right side (rest of the attributes).

Finding functional dependencies We have implemented the FUN algorithm as the *fun* function in HASKELL. This function has the following signature:

$fun :: [Attribute\ a] \to R\ a \to Relation\ b \to FDS\ a$

It receives a list of attributes that should not be considered when finding functional dependencies, a schema and a relation and computes the list of all functional dependencies induced by the data.

Filtering functional dependencies To filter "accidental" dependencies, we use the following function:

$filtering :: Weights \to FDS\ String \to R\ String \to Relation\ String \to FDS\ String$

The first argument, of type *Weights* is defined as a tuple with numbers representing the weight each of the heuristics presented in Section 2.6 have in the final result. The second argument is the set of functional dependencies to filter. The last two arguments are the schema and the relation. Using the techniques explained in Section 2.6 the function will filter out the wrong dependencies and return the relevant ones.

Normalizing functional dependencies To normalize a set of functional dependencies we use the SYNTHESIZE algorithm that is implemented by the function *synthesize*:

$synthesize :: Bool \to R\ String \to FDS\ String \to CFDS\ String$

The first argument is a boolean to encode the need of producing a set of dependencies respecting the lossless decomposition property or not. Its second argument is the relational schema since it may need the attributes to guarantee the lossless decomposition property. The third argument is the list of functional dependencies. It returns a set of normalized, up to the third normal form, compound functional dependencies.

SSFUN The algorithm *ssfun* exposed in Section 2.8, is defined in HASKELL as follows:

```
ssfun :: FilePath → R String → Relation String → IO (FDS String)
ssfun f r rl = do
  sheet ← csv2ss f ≫= return ∘ head        -- assuming that the SS has one table
  formulaAtts ← findFormulaAtts f          -- find the columns defined by formulas
  let formulaFDs = fdsFromForms sheet      -- create FDs from formulas
```

```
    $onesAtts = get1col\ (schema, relation)$    -- one value columns are separated
                                                -- create FDs for one value columns
    $onesFDs = map\ (\lambda atts \rightarrow (Atts\ [atts], Atts\ []))\ onesAtts$
    $f = fun\ (onesAtts +\!\!+ formulaAtts)\ r'\ rl'$                    -- discover FDs
                                                            -- filtering FDs
    $g = filtering\ (1,1,1,1,1)\ (f +\!\!+ formulaFDs +\!\!+ onesFDs)\ r'\ rl'$
    $h = synthesize\ True\ r\ g$                              -- normalizing FDs
    $i = selectKeys\ h$                                       -- selecting keys
  $return\ i$
```

It receives a spreadsheet (stored in a file), a relational schema and a relation and returns a set of functional dependencies. Its first task is to get the spreadsheet representation. In this simpler version we assume that the spreadsheet contains a single table.

Next, it finds the columns defined by formulas (*formulaAtts*). It then computes functional dependencies based in the columns defined by formulas (*formulaFDs*).

After this, the columns with the same value in all rows are separated from the others (*onesAtts*). These columns are then used to create functional dependencies with empty consequent (*onesFDs*). The others are used to calculate the functional dependencies induced by the spreadsheet data (*f*). Note that, the first argument of the *fun* function is the list of columns defined by formulas (*formulaAtts*) plus the columns containing the same value in all rows (*onesAtts*).

Filtering is the next step: the weights are all the same and the functional dependencies are the ones from *fun* (*f*), plus the ones from the formulas (*formulaFDs*), plus the ones from the columns with always the same value (*oneFDs*). After filtering the functional dependencies we can now normalize them using the function *synthesize*. Finally, the keys are selected and the resulting functional dependencies are returned.

8.4 Computing Models

In this section we will explain how to use the HAEXCEL libraries to devise models from a spreadsheet. We start by explaining how to generate an ER diagram.

8.4.1 Generating Entity-Relationship Diagrams

As explained in Chapter 3 we can infer an ER diagram from a spreadsheet. For this purpose, the HAEXCEL library includes two functions: *fds2ergraph* and *ss2ergraph*.

The first function receives the original schema of the spreadsheet and a set of functional dependencies (in principle, the output of *ssfun*).

$fds2ergraph :: R\ String \to FDS\ String \to String$

This function will return a string that encodes an ER diagram in the *DOT* (2011) notation. This language can be interpreted by *Graphviz* (2011) based tools. The function *fds2ergraph* makes use of two functions:

$fds2er :: FDS\ String \to ER\ Entity\ X$
$ss2eraux :: ER\ Entity\ X \to R\ String \to String$

In fact, *fds2er* does the hard work: it computes the ER diagram from the functional dependencies. The other one, *ss2eraux*, computes the string from our ER representation.

The second function creates a *DOT* file directly from a spreadsheet file.

$ss2ergraph :: FilePath \to FilePath \to IO\ ()$

This function will infer all the tables in the spreadsheets, compute the schemas and relations, use *ssfun* to compute the functional dependencies and produce the string encoding the ER diagram. In a final step it will create the output file and write on it the diagram.

8.4.2 Generating *ClassSheets*

As we saw in Chapter 3, it is possible to construct a *ClassSheet* model from a ER diagram/relational model. The function *genCS* implements this process.

$genCS :: ER\ Entity\ X \to ClassSheet$

The function receives an ER representation and returns the corresponding *ClassSheet*. The *ClassSheet* language is implemented in HASKELL as a new data type. As an example, we show its first construct here:

```
data ClassSheet = ShClass Class                   -- a class
                | ExtCl Class                 -- an expandable class
                | ClassSheet :||: ClassSheet   -- pairs of classes
                | NoSheet                      -- empty ClassSheet
```

As in the original language, a *ClassSheet* can be a class, an expandable class or a horizontal composition of two *ClassSheets*. The last constructor in necessary to represent the empty *ClassSheet*.

There are functions available for parsing and unparsing text files using this language. As an example, we list a textual representation of a simple *ClassSheet*:

```
|Income
|Item
|Item:(value=0)^
|Income:Total
        total=SUM(Item.value)
```

As the reader may have noticed, this text representation is quite similar to the original language presented in (Engels and Erwig 2005).

8.5 Edit Assistance for Spreadsheets

In Chapter 4 we have described a technique to add edit assistance to spreadsheets. In this section, we will explain how we implemented this technique so it can be used by a spreadsheet end user. To this end, we have created an *OpenOffice.org* extension. The platform chosen to implement this work was the *OpenOffice.org*, in particular the *Calc* component, since it is an open source, platform independent product. It offers some scripting languages such as *Python* or BASIC. We chose the last one since it is a very simple language and can easily be migrated to the widely used *Excel* scripting language *Visual Basic for Applications* (VBA).

As in other components of HAEXCEL, most of the work is done by the HASKELL libraries. In a first step, the BASIC is responsible for sending the spreadsheet data to the HASKELL back end through a text file in the following format: each cell is a row in a file; each row has four white space separated values: a *c* character starting a new cell, the cell column, the cell row and its content. A sample of such file is listed next:

```
c 1 1 "movieID"
c 1 2 "mv23"
c 1 3 "mv1"
```

c 1 4 `"mv21"`

...

This is received and manipulated in the HAEXCEL back end: it reads and parses the file sent by BASIC, computes the normalized set of dependencies and sends the result back to BASIC. This work is done by the function *addon*, which receives several files to read the input and write the results.

$addon :: FilePath \to FilePath \to FilePath \to FilePath \to FilePath \to IO\ ()$

8.5.1 Bidirectional Auto-Completion

To accomplish the bidirectional auto-completion described in Chapter 4, it is necessary to generate *combo boxes*, that is, a button that when clicked shows a list of possible selection values. It also needs to generate some additional formulas. We describe these two steps next.

Generating Combo Boxes Remember from Section 4.3 that for each functional dependency $a_1, ..., a_n \rightharpoonup c_1, ..., c_m$, are generated $n + m$ *combo boxes*, that is, each attribute/column has its own combo box. Let r be the new row, the row that we are introducing data now, and *minr* be the first row with data. Then, to the antecedent attributes/columns, the following combo boxes are generated:

$$\begin{aligned}
&\forall\, c \in \{a_1, ..., a_n\}:\\
&\quad S\,(c,r) = combobox := \{\, linked\_cell := (c,r);\\
&\qquad\qquad source\_cells := (c, minr) : (c, r-1);\\
&\qquad\qquad backgoundColor := green \,\}
\end{aligned}$$

For the consequent attributes/columns, the formula is:

$$\begin{aligned}
&\forall\, c \in \{c_1, ..., c_m\}:\\
&\quad S\,(c,r) = combobox := \{\, linked\_cell := (c,r);\\
&\qquad\qquad source\_cells := (c, minr) : (c, r-1);\\
&\qquad\qquad backgoundColor := red \,\}
\end{aligned}$$

The combo box is linked to the cell under it, *linked_cell*. The source cells, that is, its possible choices, are all the previous cells with data in the same column, *source_cells*. The combo boxes representing antecedents have green background (*backgoundColor*) and the others red.

To send this information back to the spreadsheet, we use a text file. Each row in this file is used to create a combo box, where the first value is the column to place it, the second (third) value is the lower (upper) bound of its cell range.

Note that there are two different sets of combo boxes: one to the antecedent attributes/columns and another to the consequent attributes/columns. These two sets must be separated since they will have different characteristics in the spreadsheet.

For each row in the referred files, BASIC generates a combo box using the routine *createCB*. The main function tests if the file with the HAEXCEL result was already created and then reads each line, parses it and sends to *createCB* the column and the row to draw the combo box, the start and end rows of the data, the combo box position (this value is a pair of drawing coordinates) and the background color of the combo box. If the back end is not finished yet, it waits half of a second and tries again. When it finishes, it deletes the file. The *createCB* function starts by creating and positioning a drawing zone to place the combo box. Then, it creates the combo box itself. After that, it creates the cell range and the linked cell.

$$\textit{Sub createCB}\ (\textit{col as int}, \textit{row as int}, \textit{minr as int}, \textit{maxr as int}, \textit{px as int}, \textit{py as int})$$

The routine *createCB* receives the column *col* and the row *row* where to insert the combo box, the interval *minr–maxr* of the choice values that the combo box will show and the coordinates *px* and *py* where to draw the combo box. As a result it will draw the combo box with the intended attributes.

Formulas Auto-completion cannot be accomplished just by the use of combo boxes: we need extra formulas in the spreadsheet too. After choosing a value in an antecedent column, the other columns of the same dependency must be informed of such a change. Remember the formula introduced in Section 4.3:

$$
\begin{array}{l}
\forall\, c \in \{c_1, ..., c_m\}, \forall\, r \in \{minr, ..., maxr\}: \\
\quad S\,(c,r) = \textbf{if}\;(\textbf{if}\;(\textbf{isna}\;(\textbf{vlookup}\;((a_1,r),(a_1,r1):(c,r-1),r-a_1+1,0)), \\
\qquad\qquad\qquad "", \\
\qquad\qquad\qquad \textbf{vlookup}\;((a_1,r),(a_1,r1):(c,r-1),r-a_1+1,0)) \\
\qquad\qquad\quad == \\
\qquad\qquad\quad \textbf{if}\;(\textbf{isna}\;(\textbf{vlookup}\;((a_2,r),(a_2,r1):(c,r-1),r-a_2+1,0)), \\
\qquad\qquad\qquad "", \\
\qquad\qquad\qquad \textbf{vlookup}\;((a_2,r),(a_2,r1):(c,r-1),r-a_2+1,0)) \\
\qquad\qquad\quad == ... ==
\end{array}
$$

$$\begin{array}{l}
\textbf{if}\ (\textbf{isna}\ (\textbf{vlookup}\ ((a_n, r), (a_n, r1) : (c, r-1), r - a_n + 1, 0)), \\
\quad "", \\
\quad \textbf{vlookup}\ ((a_n, r), (a_n, r1) : (c, r-1), r - a_n + 1, 0)), \\
\textbf{vlookup}\ ((a_1, r), (a_1, r1) : (c, r-1), r - a_1 + 1, 0), \\
"")
\end{array}$$

For each consequent combo box, the HASKELL back end will compute a string containing the formula which is then passed to the BASIC front end throw a file. Each formula is accompanied by the row and column where to put it. The BASIC front end will copy it to the correct cell.

8.5.2 Safe Deletion

Deletion is a concern when dealing with data. One can easily destroy data that is not represented elsewhere. For instance, in our running example of Chapter 4, Figure 4.2, if the user deletes the third row, all the information about the movie `The OH in Ohio` will disappear! Since this is a renting system, probably the user only wants to delete the renting transaction.

As explained in Section 4.5, to avoid this problem we introduce a new a *button* for each row with data. This button tests if there is risk of loosing data, by checking for each relational table, if there is other row that contains the same information that is being deleted. If it is the case, the row is deleted, otherwise the system asks the user for confirmation.

The following formula assigns to each row with data an a button with an event listener that checks for the explained deletion anomalies. Let *col* be the column after the data and $a_1, ..., a_n \rightharpoonup c_1, ..., c_m$ be a functional dependency.

$$\begin{array}{l}
\forall\, r \in \{minr, ..., maxr\}, ants \in \{a_1, ..., a_n\} : \\
\quad S\ (col, r) = delete := \{onClick := delete\ (r, ants); \}
\end{array}$$

The delete routine receives the row that is assigned to it and the columns that are antecedents.

$$Sub\ delete\ (row\ as\ integer, cols)$$

It will check if the values in that row are unique. If this is the case, it will trigger a message to user warning him. The user then decides to continue or to stop the deletion.

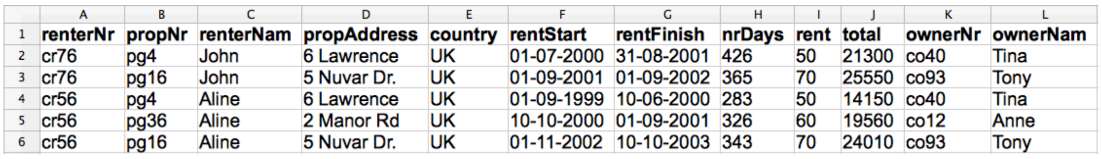
8.6 Migration of Spreadsheets

In this section we present the HAEXCEL component capable of refactoring a spreadsheet: based on a relational schema, it can produce a new spreadsheet with the same data as the original, but normalized as a relational database. Moreover, this component can generate SQL scripts to create and populate a normalized database.

8.6.1 From Spreadsheets to Databases

As we said, this system is intended to migrate spreadsheets to normalized spreadsheets (or databases). In fact, we can calculate a normalized schema based on a spreadsheet, but we cannot make the system transform the original spreadsheet into a new one based on the computed schema. To overcome this problem, we use a little tweak: we calculate the type of the normalized database based on the relational schema, rearrange the data to fit this new model, and then use the transformation rule (*ss2rdb*) defined to migrate between databases and spreadsheets. Remember that the rule is an isomorphism and so it can be freely used in both directions.

To make our tool easier to understand, remember the spreadsheet example used in Chapter 5 shown in Figure 8.2:



	A	B	C	D	E	F	G	H	I	J	K	L
1	**renterNr**	**propNr**	**renterNam**	**propAddress**	**country**	**rentStart**	**rentFinish**	**nrDays**	**rent**	**total**	**ownerNr**	**ownerNam**
2	cr76	pg4	John	6 Lawrence	UK	01-07-2000	31-08-2001	426	50	21300	co40	Tina
3	cr76	pg16	John	5 Nuvar Dr.	UK	01-09-2001	01-09-2002	365	70	25550	co93	Tony
4	cr56	pg4	Aline	6 Lawrence	UK	01-09-1999	10-06-2000	283	50	14150	co40	Tina
5	cr56	pg36	Aline	2 Manor Rd	UK	10-10-2000	01-09-2001	326	60	19560	co12	Anne
6	cr56	pg16	Aline	5 Nuvar Dr.	UK	01-11-2002	10-10-2003	343	70	24010	co93	Tony

Figure 8.2: A spreadsheet representing a property renting system.

Remember also that, based on the data in our example spreadsheet and using the techniques presented in Chapter 2, we can discover the following relational database schema:

$$
\begin{array}{ll}
Country & (\underline{country}) \\
Renter & (\underline{renterNr}, renterNam) \\
Owner & (\underline{ownerNr}, ownerNm) \\
Property & (\underline{propNr}, propAddress, rent, \#ownerNr) \\
<Renting> & (\underline{\#renterNr, \#propNr, \#country, rentStart, rentFinish}, nrDays, total)
\end{array}
$$

Using the function *fds2type*, we can transform the above schema into the following representation in the 2LT framework:

$$
\begin{array}{l}
\texttt{"country"} \rightharpoonup (One \times Int) \times \\
\quad \texttt{"renterNr"} \rightharpoonup (\texttt{"renterNam"} \times Int) \times \\
\quad\quad \texttt{"ownerNr"} \rightharpoonup (\texttt{"ownerNam"} \times Int) \times \\
\quad\quad\quad \texttt{"propNr"} \rightharpoonup (\texttt{"propAddress"} \times \texttt{"rent"} \times \texttt{"ownerNr"} \times Int) \times \\
\quad\quad\quad\quad \texttt{"renterNr"} \times \texttt{"propNr"} \times \texttt{"country"} \times \texttt{"rentStart"} \times \\
\quad\quad\quad\quad \texttt{"rentFinish"} \rightharpoonup (\texttt{"nrDays"} \times \texttt{"total"} \times Int)
\end{array}
$$

Each table is represented by a map from the primary key attributes to the non-key ones. For example, the table *Renter* is transformed into the map from the renter number to its name. If a table has only key attributes, the map is from the key attributes to the *nil* object (*One*).

In fact, the above type is not complete because the foreign keys are still missing, which are represented by the four invariants show next:

$$
\begin{array}{l}
inv1 = \pi_{ownerNr} \circ \rho \circ \pi_{Property} \subseteq_s \delta \circ \pi_{Owner} \\
inv2 = \pi_{renterNr} \circ \delta \circ \pi_{Renting} \subseteq_s \delta \circ \pi_{Renter} \\
inv3 = \pi_{propNr} \circ \delta \circ \pi_{Renting} \subseteq_s \delta \circ \pi_{Property} \\
inv4 = \pi_{country} \circ \delta \circ \pi_{Renting} \subseteq_s \delta \circ \pi_{Country}
\end{array}
$$

To understand these invariants, we must understand some functions: π_A represents the projection of the map representing the table A; π_a represents the projection of the attribute a; δ represents the domain of a map and ρ its range.

The invariant *inv1* represents the foreign key from property to owner. It can be read as follows: the result from projecting our database by its property table, taking its range and then projecting the owner number, must be included in the domain of the projection of the owner table. This is one way of expressing a foreign key. The others work in an analogous way. Apart from these invariants, there is another per table guaranteeing that the integer is greater than zero, as explained in Section 5.3.

8.6.2 From Databases to Spreadsheets

Since we now have the type of the database, we can use the strategy *direct_ss2rdb* (defined in Section 5.4.4) to migrate our data to a new refactored spreadsheet. We show the migrated data next:

$$
\begin{aligned}
&([(UK,())],\\
&([(cr76,John),\\
&\quad(cr56,Aline)],\\
&([(co40,Tina),\\
&\quad(co93,Tony),\\
&\quad(co12,Anne)],\\
&([(pg4,(6\ Lawrence,(50,co40))),\\
&\quad(pg16,(5\ Nuvar\ Dr.,(70,co93))),\\
&\quad(pg36,(2\ Manor\ Rd,(60,co12)))],\\
&[\ ((cr76,(pg4,(UK,(01-07-2000,31-08-2001)))),(426,21300)),\\
&\quad((cr76,(pg16,(UK,(01-09-2001,01-09-2002)))),(365,25550)),\\
&\quad((cr56,(pg4,(UK,(01-09-1999,10-06-2000)))),(283,14150)),\\
&\quad((cr56,(pg36,(UK,(10-10-2000,01-09-2001)))),(326,19560)),\\
&\quad((cr56,(pg16,(UK,(01-11-2002,10-10-2003)))),(343,24010))]))))
\end{aligned}
$$

It is composed by four lists of data, one per table. Each element of the lists is a pair: in the first component it contains the data of the key columns and in the second the non-key attributes.

To make the use of this tool more easy, we implemented the function *ss2SS*.

$$ss2SS :: FilePath \to FilePath \to IO\ ()$$

It receives a file with the original spreadsheet, and a second file to save the new refactored and normalized one. All the steps presented before are done in an automatic way using this function.

To make this feature available to a larger audience, we created an online form where the user can submit a spreadsheet and receives a new refactored spreadsheet.

8.6.3 Generating Databases

The HAEXCEL libraries also generate SQL code which creates the database according to the derived schema. This is basically a simple SQL `create` instruction based on the relational schema. Furthermore, it produces SQL code to insert the migrated data in the database, and, again, this corresponds to a SQL `insert` instruction with the migrated data as argument. Because some values of the spreadsheet are defined through formulas, we generate also SQL triggers, that model the spreadsheet formulas, which are used to update the database and guarantee its integrity. We can also generate functions to compute such values. Next, we present the trigger induced by the two formulas of our running example:

```
create trigger ssformulas before insert on tbl
  for each row begin
    set new.nrDays = new.rentFinish - new.rentStart;
    set new.total = new.rent * new.nrDays;
  end;
```

As an example, we show the SQL code to create and populate the client table.

```
create table tbl (renterNr varchar(256),
                  renterNam varchar(256),
                  primary key (renterNr));

insert into tbl values ("cr76","John"),("cr56","Aline");
```

The generated script can be imported by any relational database schema and create and populate the database.

As in the generation of a refactored spreadsheet, an online version of this feature is available. Given a spreadsheet, the tool will calculate the relational schema and generate a SQL script to create and populate a database.

8.7 Evolution of Spreadsheets

In this section we will present the last component of HAEXCEL: safe evolution of spreadsheets . As we said before, it is difficult to keep changes in models and instances synchronized. To solve this problem for spreadsheets we reuse an existing framework, 2LT, for coupled software transformations. In 2LT it is possible to design rules to evolve models and instances in general. Unfortunately, the existing implementation does not work for spreadsheets. So, our first step was to improve the framework so it could support spreadsheet models and their instances.

To encode spreadsheet models, we based our work on *ClassSheets*. We remember now the combinators created to encode these models, as presented in Section 6.3 next:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Type}\ a\ \textbf{where} & & \\
\quad \ldots & & \\
\quad \mathit{Value} & :: \mathit{Value} \to \mathit{Type}\ \mathit{Value} & \text{-- plain value} \\
 & & \text{-- references} \\
\quad \mathit{Ref} & :: \mathit{Type}\ b \to \mathit{PF}\ (a \to \mathit{RefCell}) \to \mathit{PF}\ (a \to b) \to \mathit{Type}\ a \to \mathit{Type}\ a & \\
\quad \mathit{RefCell} & :: \mathit{Type}\ \mathit{RefCell} & \text{-- reference cell} \\
\quad \mathit{Formula} & :: \mathit{Formula} \to \mathit{Type}\ \mathit{Formula} & \text{-- formulas} \\
\quad \mathit{LabelB} & :: \mathit{String} \to \mathit{Type}\ \mathit{LabelB} & \text{-- block label} \\
\quad \cdot = \cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a,b) & \text{-- attributes} \\
\quad \cdot \shortmid \cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a,b) & \text{-- block horizontal composition} \\
\quad \cdot\,\hat{}\,\cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a,b) & \text{-- block vertical composition} \\
\quad \mathit{EmptyB} & :: \mathit{Type}\ \mathit{EmptyB} & \text{-- empty block} \\
\quad \underline{\cdot} & :: \mathit{String} \to \mathit{Type}\ \mathit{HorH} & \text{-- horizontal class label} \\
\quad \mid \cdot & :: \mathit{String} \to \mathit{Type}\ \mathit{VerV} & \text{-- vertical class label} \\
\quad \mid \underline{\cdot} & :: \mathit{String} \to \mathit{Type}\ \mathit{Square} & \text{-- square class label} \\
\quad \mathit{LabRel} & :: \mathit{String} \to \mathit{Type}\ \mathit{LabS} & \text{-- relation class} \\
\quad \cdot : \cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a,b) & \text{-- labeled class} \\
\quad \cdot : (\cdot)^{\downarrow} & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a,[b]) & \text{-- labeled expandable class} \\
\quad \cdot\,\hat{}\,\cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a,b) & \text{-- class vertical composition} \\
\quad \mathit{SheetC} & :: \mathit{Type}\ a \to \mathit{Type}\ (\mathit{SheetC}\ a) & \text{-- sheet class} \\
\quad \cdot^{\to} & :: \mathit{Type}\ a \to \mathit{Type}\ [a] & \text{-- sheet expandable class} \\
\quad \cdot \shortmid \cdot & :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a,b) & \text{-- sheet horizontal composition} \\
\quad \mathit{EmptyS} & :: \mathit{Type}\ \mathit{EmptyS} & \text{-- empty sheet}
\end{array}
$$

As an example of how to use this type, we show how to encode the renter table of the example given in the previous section:

$$
\begin{array}{l}
\mathit{renter} = \underline{\texttt{"Renter"}} : \texttt{"Renter Nr"} \shortmid \texttt{"Renter Nam"}\,\hat{} \\
\qquad\qquad\qquad (\texttt{"renterNr"} = \texttt{"norenternr"} \shortmid \texttt{"renterNam"} = \\
\texttt{"norenternam"})^{\downarrow}
\end{array}
$$

The class *Renter* is defined as a labeled expandable class, $\cdot : (\cdot)^{\downarrow}$, with label `"Renter"` and is composed by a vertical block composition, $\cdot\,\hat{}\,\cdot$, where the top block contains the labels for the columns, `"Renter Nr"` and `"Renter Nam"`. The bottom part is a horizontal block composition, $\cdot \shortmid \cdot$, that on the left has the renter's number default value, `"norenternr"`, and on the right has the default renter's name, `"norenternam"`.

The values from the previous section for renters are encoded as follows:

```
renterData = ("Renter", (("Renter Nr", "Renter Nam"),
                        [("cr76", "John"),
                         ("cr56", "Aline")]))
```

Suppose now we want to add the phone number to the renters. The rule to add a new column could be defined as follows:

```
insertPhone = insertColIn "renterPhone" (FValue ("norenterphon"))
```

This function will create a vertical block composition with label *renterPhone* (on top) and default value "norenterphon" (on the bottom) and will try to insert it in the existing renter model. In fact, we know that it will fail because it tries to insert a block after another block, but it receives a class. To correctly execute this function, we need to use the combinator *once*: it will try to find one place to apply the given rule in the received model. So, its correct usage would be

```
insertPhone2 =
  once (insertColIn "renterPhone" (FValue ("norenterphon")))
```

To apply the forward transformation of such rule, we need to define the new model, as shown next:

```
newRenterModel = "Renter" :
  "Renter Nr" | "Renter Name" | "Renter Phone"^
  ("renterNr" = "norenternr" | "renterNam" = "norenternam" |
    "renterPhon" = "norenterphon")↓
```

The HASKELL code to apply the transformation is as follows:

```
go = forth (fromJust ((insertPhone2 ⊳ normalize) renter))
           newRenterModel
           renterData
```

Note that, the rule *insertPhone2* is not applied alone, but followed by the *normalize* rule. This is necessary because the output of *insertColIn* has a bad format, that is, it does not represent the spreadsheet we want to have. The layout rule *normalize* will fix this.

Now, if we apply the forward transformation of the rule to the above renter's data, we get the following new data:

```
newRenterData =
  ("Renter",(("Renter Nr",("Renter Nam","Renter Phon")),
             [("cr76",("John","norenterphon")),
              ("cr56",("Aline","norenterphon"))]))
```

As the reader may have noticed, the component of the tool just described is very low level, which is not ideal for end users. As future work, we intend to create an extension for a spreadsheet system to make this available for end users. All this machinery would be replaced by buttons to add columns, normalize models, etc. These buttons would run this HASKELL functions on the background and change the spreadsheet according to their output. One can imagine a spreadsheet with two sheet, one for the model, that would need to be embedded in the spreadsheet language, and another with an instance of such model. Changes in the model would be automatically reflected on the instance.

8.8 Conclusions

In this chapter we have presented in detail the framework we developed, HAEXCEL. This framework has several components, each of them implementing part of the techniques introduced in this thesis. Although the framework already makes available all the techniques, we would like to make them more usable. In particular, we would like to fully integrate it with *OpenOffice.org* or *Excel* and also to make it available through a web page so users could upload a spreadsheet and get back all the spreadsheets we can generate. The HAEXCEL framework is available from the homepage of the author: `http://www.di.uminho.pt/~jacome`.

Chapter 9

Conclusions

Summary

In this thesis we have developed a number of techniques for model-based engineering of spreadsheets that are intended to help users to avoid some of the drawbacks of spreadsheets. In this chapter, we look back at the techniques we developed and assess to what extent they provide answers to the research questions we have proposed in Chapter 1.

9.1 Contributions

The overall contribution of this thesis is the development of theories and tools for extracting models from spreadsheets and using these models to make it more safe to edit, refactor, and migrate spreadsheets. In more detail, the contributions of the thesis are the following:

Inference of functional dependencies for spreadsheets (Chapter 2) Functional dependencies are widely used in the context of relational databases and their adoption in other areas is not straightforward. We studied and presented techniques to automatically infer and reason about functional dependencies in the context of spreadsheets.

Inference of relational schemas for spreadsheets (Chapter 3) We explored the idiosyncrasies of spreadsheets to automatically derive relational database schemas for spreadsheets. In fact, we use the functional dependencies inferred from the data as the basis for this computation. These schemas can be used, for example, to migrate spreadsheets.

Inference of *ClassSheets* for spreadsheets (Chapter 3) Although relational schemas are very expressive for databases, they cannot capture, for example, the layout of a spreadsheet. Even though they are enough for a migration to databases, for tasks like evolution of spreadsheets, more expressive models are necessary. To this end and from the inferred functional dependencies we were able to automatically compute *ClassSheet* models. This is particularly important for existing spreadsheets since it is difficult for an end user to define a specification for an already created spreadsheet.

Mapping *ClassSheets* into UML class diagrams (Chapter 3) Since *ClassSheets* are based on the OO paradigm, a smooth transition between *ClassSheets* and UML class diagrams is possible and we have done it. From these class diagrams, migrations to other paradigms are now possible.

Generation of edit assistance (Chapter 4) Using functional dependencies we were able to infer edit assistance for spreadsheets. This includes, for example, the auto-completion of some columns in a spreadsheet based on the values of other columns. Using this assistance, users can insert data in a spreadsheet with less effort since part of the data is automatically introduced.

Migration of spreadsheets to databases and vice versa (Chapter 5) We calculated the formal relationship between spreadsheet models and relational database schemas. Rules to migrate between these two formalisms were designed.

Techniques to generate refactored spreadsheets (Chapter 5) Based on a relational schema we were able to produce a new spreadsheet that is more organized, from a data point of view, than the original one and thus better for handling data. In fact, this new spreadsheet is normalized and so it prevents data inconsistencies, data corruption and data redundancy.

Improved the 2LT framework to support spreadsheets (Chapter 6) We improved the *Two Level Transformation* platform to support spreadsheet models (based on *ClassSheets*). We also presented a technique to encode spreadsheet instances under this framework.

Rules for evolution and refactoring of spreadsheets (Chapter 6) On top of the 2LT framework we developed a series of common evolution steps for spreadsheets. These steps include insertion of a column in each instance of a model or the factorization of common parts of the spreadsheet.

Empirical validation (Chapter 7) We validated several of the techniques presented in the previous chapters in a study with human end users proving the validity of our work. In particular, we proved that, in certain conditions, users make fewer errors and are faster when using spreadsheets based on our models.

The HAEXCEL framework (Chapter 8) All the techniques presented in this thesis are implemented and available under an open source framework termed HAEXCEL. This framework includes online and batch tools, reusable HASKELL libraries and extensions to spreadsheet environments. This framework is available from the homepage of the author: `http://www.di.uminho.pt/~jacome`.

9.2 Answers to the Research Questions

In this section we discuss the work presented in this thesis answering the research questions posed in Chapter 1.

RQ1: Can we automatically infer the implicit logic of a spreadsheet and produce a specification or model describing it?

We used a strong concept from relational databases theory, functional dependencies, to deduce relationships between the data in a spreadsheet, as explained in Chapter 2. These dependencies are the building blocks for the specifications that we can automatically infer from spreadsheets. A relational schema, a *ClassSheet* model and a UML class diagram can also be inferred as explained in Chapter 3. Therefore, we can say that it is possible to infer the business logic of a spreadsheet from its content. In fact, we are able to derive three different models: one more related to the data itself, the relational schema, one more specific to spreadsheets allowing its full specification, the *ClassSheet*, and one more general, the UML class diagram.

In fact, we did even more: the migration between relational databases and spreadsheets and vice versa was discussed in Chapter 5. The relational schema inferred for a spreadsheet can be used to migrate that spreadsheet to a relational database. Since we used the 2LT framework to do this task, the inverse process is also possible.

Furthermore, we can also generate SQL scripts. This generation is quite useful because one can very easily generate a script that can be used to create and populate a database with the same data contained in the spreadsheets.

Although all the available features are automatic, in some cases, results may not always be perfect. For example, in Chapter 3, the inference of names for relations was not very good and we actually changed them by hand. Furthermore, our techniques work better for spreadsheets that are data intensive oriented.

RQ2: Can we use these specifications/models to improve spreadsheet environments in such a way that end users commit fewer errors?

The models inferred can be used to generate new spreadsheets. One kind of spreadsheet contains edit assistance for end users, as discussed in Chapter 4. Helping end users editing a spreadsheet, for example, auto completing some columns, is one of the features of this spreadsheet.

Another kind of spreadsheet that we can generate from the models is one that is refactored in such a way that it enforces the usage of a relational schema and thus eliminates update and delete anomalies, as explained in Chapter 5.

The former spreadsheet should be used when the user wants to keep the original layout of the spreadsheet, but still wants some help. The latter spreadsheet can be used in cases without this constraint. If the user is willing to have a new format, the refactored spreadsheet is more appropriate.

These two spreadsheets were subject of an empirical study with human end users and proved, in some situations, to help the end users to commit fewer errors, as discussed in Chapter 7. Moreover, these models can be used for safe evolution of spreadsheets as proposed in Chapter 6. The editing of spreadsheets is a very error prone task, specially when the underlying model also needs to be changed. Allowing safe editing of models and instances guarantees that the spreadsheets always conform to the specification. Thus, we can say that models/specifications can be used to help spreadsheet end users to make fewer mistakes.

In fact, our techniques not only improved end-users' error rate, but they also allow users to manipulate spreadsheets faster. Our models help end users to insert, edit and query spreadsheets faster than before, as discussed in Chapter 7.

RQ3: To which extent can we create specifications for spreadsheets and improve them in a non-invasive way, that is, not disturbing the end user?

All the techniques developed during this study are automatic, that is, all the inference of functional dependencies and models and the generation of new spreadsheets is completely autonomous. This is quite important since it does not require the user to learn anything new or to explicitly give new knowledge to the system. We believe that this kind of techniques are easier to adopt by users. Therefore, we can say that it is possible to infer models for spreadsheets and use them to make spreadsheets safer in a non-invasive way, that is, without interfering or interrupting the user's work flow.

The techniques we developed have all been implemented as part of a framework, HAEXCEL. It is an open source HASKELL based tool composed by online tools, batch tools that can be compiled for any platform (supported by HASKELL), reusable HASKELL libraries and *OpenOffice.org* extensions. Since it is open source, it can be reused for other projects.

9.3 Future Work

In this section we discuss some interesting future work directions related to the work presented in this thesis.

Spreadsheets meet the Web

Nowadays, more and more applications are being ported to work online, as web applications. This trend is also being adopted by spreadsheet applications makers, such as Microsoft or Google. Unfortunately, our tools cannot work directly with this new way of using spreadsheets. Nevertheless, we believe our techniques still apply, although probably not without changes, given that the interaction with the users is slightly different. It would be of great interest to study the applicability of the work we presented in this thesis to this web world.

Furthermore, this new way of using spreadsheets makes it easier to do collaborative work. This poses several problems, such as, for example, the synchronization of spreadsheets. Imagine a situation in a company where one department works on part of a spreadsheet, a view, and another department on a second view. The administration probably needs to view the complete spreadsheet. The synchronization of two or more software artifacts is not an easy task (Antkiewicz and Czarnecki 2008). In particular, the synchronization of spreadsheets stands in the same difficulty standard. Techniques to achieve this purpose should be studied and proposed.

Spreadsheets meet MDE Again

In this thesis we have provided a series of theories, techniques and tools based on models to help end users when using spreadsheets. We have shown that, in some cases, this MDE approach can help users being more effective and efficient. Despite this result, we can also see that MDE is a very complex concept and it is very difficult for end users to understand it and take full advantage of it. Therefore, we believe that more research is necessary to make MDE techniques available in a broader way for the end-users community. How to effectively construct tools for end users is still an open issue and, given the more and more relevance of non-professional programmer, deserves further studies.

The work we presented was done mostly thinking of end users and how to help them. The results achieved could have been different if professional users were the main target. The idiosyncrasies of spreadsheets like, for example, the layout restrictions or the non-separation between data and computations, pose a particular challenge. Although software engineering techniques work well for programming languages for professionals, to assess their use by professionals in spreadsheets is an interesting research path. To improve or adapt these techniques for this realm should be an interesting work to do, possibly with good results.

Spreadsheets meet Bidirectional Transformations

An interesting area is to explore bidirectional transformations between a model of a spreadsheet and its instances. One can imagine a situation where a professional programmer would define a formal model for a spreadsheet using, for example, *ClassSheet* models. On the other hand, end users would have their own spreadsheet to work on. In our work we have defined ways of changing the models and instances at the same time, but it would be interesting to define techniques to propagate changes on models to the spreadsheet instances. Moreover, to reflect changes in the spreadsheet in the model could also be an objective. In fact, the bidirectional interaction between the model and the instances should be the goal.

A final proposal of future work is to study debugging of spreadsheets, also using bidirectional transformations. To this end, we can observe that spreadsheets have two possible views: one view contemplates the formulas and another focuses on the results of these formulas. One of the biggest sources of errors in spreadsheets is the replacement of a formula by a value, involuntarily or to have the correct value in such cell. In fact, this could be useful: if this change was back-propagated to the rest of the spreadsheet, the user could have the perception of the reason why that result is wrong. In fact, a similar approach is followed in (Abraham and Erwig 2007a). Bidirectional transformations (Czarnecki *et al.* 2009) could be used to achieve this goal. A transformation between the entire spreadsheet to the result of its formulas and an inverse could be used to understand the impact of changes in certain cell/formulas. The proposed approaches in Chapters 5 and 6 for bidirectional transformations should give some insight to solve this problem.

Bibliography

Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 165–172, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8696-5. **Cited** on pages 4 and 6.

Robin Abraham and Martin Erwig. Inferring templates from spreadsheets. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 182–191, New York, NY, USA, 2006a. ACM. ISBN 1-59593-375-1. **Cited** on pages 7, 44, 70, 90, 114, 133 and 134.

Robin Abraham and Martin Erwig. Type inference for spreadsheets. In Annalisa Bossi and Michael J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, pages 73–84. ACM, 2006b. ISBN 1-59593-388-3. **Cited** on page 90.

Robin Abraham and Martin Erwig. AutoTest: A tool for automatic test case generation in spreadsheets. In *VL/HCC*, pages 43–50. IEEE Computer Society, 2006c. ISBN 0-7695-2586-5. **Cited** on page 5.

Robin Abraham and Martin Erwig. GoalDebug: A spreadsheet debugger for end users. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 251–260, Washington, DC, USA, 2007a. IEEE Computer Society. ISBN 0-7695-2828-7. **Cited** on page 179.

Robin Abraham and Martin Erwig. UCheck: A spreadsheet type checker for end users. *Journal of Visual Languages and Computing*, 18(1):71–95, 2007b. ISSN 1045-926X. **Cited** on pages 39, 85 and 154.

Robin Abraham, Martin Erwig, Steve Kollmansberger, and Ethan Seifert. Visual specifications of correct spreadsheets. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 189–196, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2443-5. **Cited** on pages 43, 70 and 114.

Yanif Ahmad, Tudor Antoniu, Sharon Goldwater, and Shriram Krishnamurthi. A type system for statically detecting spreadsheet errors. In *ASE '03: Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 174–183, October 2003. **Cited** on page 6.

Reda Alhajj. Extracting the extended entity-relationship model from a legacy relational database. *Information Systems*, 28(6):597–618, 2003. ISSN 0306-4379. **Cited** on pages 21, 50, 54 and 138.

Tiago L. Alves, Paulo F. Silva, and Joost Visser. Constraint-aware Schema Transformation. In *The Ninth International Workshop on Rule-Based Programming*, 2008. **Cited** on pages 16, 96, 99 and 118.

Michał Antkiewicz and Krzysztof Czarnecki. Design space of heterogeneous synchronization. In *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 3–46, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88642-6. **Cited** on page 178.

Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. Validating the unit correctness of spreadsheet programs. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 439–448, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. **Cited** on page 6.

Paolo Atzeni and Valeria De Antonellis. *Relational database theory*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993. ISBN 0-8053-0249-2. **Cited** on page 22.

Laura Beckwith, Jácome Cunha, Jo ao Paulo Fernandes, and Joo Saraiva. End-users productivity in model-based spreadsheets: An empirical study. In *Proceedings of* *the Third International Symposium on End-User Development*, IS-EUD '11, pages 282–288, 2011a. **Cited** on page 6.

Laura Beckwith, Jácome Cunha, João Paulo Fernandes, and João Saraiva. An empirical study on end-users productivity using model-based spreadsheets. In Simon Thorne and Grenville Croll, editors, *Proceedings of the European Spreadsheet Risks Interest Group*, EuSpRIG '11, pages 87–100, July 2011b. ISBN 978-0-9566256-9-4. **Cited** on page 6.

Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. Coupled schema transformation and data conversion for XML and SQL. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 290–304. Springer Berlin / Heidelberg, 2007. **Cited** on pages 7 and 73.

Jean Bézivin. Model driven engineering: An emerging technical space. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer Berlin / Heidelberg, 2006. **Cited** on page 13.

Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-oriented analysis and design with applications*. Addison-Wesley Professional, third edition, 2007. ISBN 9780201895513. **Cited** on pages 3 and 4.

David J. Bookbinder. *The Lotus guide to 1-2-3: Release 3*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0-201-15038-7. **Cited** on page 2.

Margaret Burnett, Andrei Sheretov, Bing Ren, and Gregg Rothermel. Testing homogeneous spreadsheet grids with the "What You See Is What You Test" methodology. *IEEE Transactions on Software Engineering*, 28(6):576–594, Jun 2002. ISSN 0098-5589. **Cited** on page 6.

Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 93–103, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. **Cited** on page 6.

Mary V. Campbell. *Using Excel.* Que Corp., Indianapolis, IN, USA, 1985. ISBN 0880222093. **Cited** on pages 2 and 78.

Peter Pin-Shan Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, March 1976. ISSN 0362-5915. **Cited** on pages 26 and 56.

Michael J. Coblenz, Andrew J. Ko, and Brad A. Myers. Using objects of measurement to detect spreadsheet errors. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 314–316, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2443-5. **Cited** on page 6.

Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13:377–387, June 1970. ISSN 0001-0782. **Cited** on pages 8 and 9.

Edgar F. Codd. *Further Normalization of the Data Base Relational Model.* In Rustin (ed). Englewood Cliffs, NJ: Prentice-Hall, 33–64, 1972. **Cited** on pages 27 and 36.

Thomas M. Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0201708574. **Cited** on pages 35, 92 and 138.

Grenville J. Croll. The importance and criticality of spreadsheets in the city of London. *CoRR*, abs/0709.4063, 2007. **Cited** on page 5.

Grenville J. Croll. Spreadsheets and the financial collapse. *CoRR*, abs/0908.4420, 2009. **Cited** on page 5.

Alcino Cunha and Joost Visser. Strongly typed rewriting for coupled software transformation. *Electronic Notes on Theoretical Computer Science*, 174:17–34, April 2007a. ISSN 1571-0661. **Cited** on pages 96, 112 and 118.

Alcino Cunha and Joost Visser. Transformation of structure-shy programs: Applied to XPath queries and strategic functions. In G. Ramalingam and Eelco Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, pages 11–20. ACM, 2007b. ISBN 978-1-59593-620-2. **Cited** on pages 96 and 118.

Alcino Cunha, José N. Oliveira, and Joost Visser. Type-safe two-level data transformation. In J. Misra *et al.*, editors, *Proceedings of the 14th International Symposium on Formal Methods Europe*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006. **Cited** on pages 16, 96, 97, 99, 102, 114 and 118.

Jácome Cunha, João Saraiva, and Joost Visser. From spreadsheets to relational databases and back. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 179–188, New York, NY, USA, 2009a. ACM. ISBN 978-1-60558-327-3. **Cited** on pages 6 and 133.

Jácome Cunha, João Saraiva, and Joost Visser. Discovery-based edit assistance for spreadsheets. In *VLHCC '09: Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 233–237, Washington, DC, USA, 2009b. IEEE Computer Society. ISBN 978-1-4244-4876-0. **Cited** on pages 6, 114 and 133.

Jácome Cunha, Martin Erwig, and João Saraiva. Automatically inferring ClassSheet models from spreadsheets. In *VLHCC '10: Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 93–100, Washington, DC, USA, 2010. IEEE Computer Society. **Cited** on pages 6, 114 and 133.

Jácome Cunha, Joost Visser, Tiago Alves, and João Saraiva. Type-safe evolution of spreadsheets. In *FASE '11: Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011*, pages 186–201, Berlin, Heidelberg, 2011. Springer-Verlag. **Cited** on page 6.

Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT '09: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02407-8. **Cited** on page 179.

Christopher J. Date. *A guide to the SQL standard.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-05777-8. **Cited** on pages 11 and 95.

Islay Davies, Peter Green, Michael Rosemann, Marta Indulska, and Stan Gallo. How do practitioners use conceptual modeling in practice? *Data & Knowledge Engineering*, 58:358–380, September 2006. ISSN 0169-023X. **Cited** on page 56.

Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970. **Cited** on page 4.

DOT. The DOT Language. `http://www.graphviz.org/doc/info/lang.html`, 2011. **Cited** on page 160.

Gregor Engels and Martin Erwig. ClassSheets: Automatic generation of spreadsheet applications from object-oriented specifications. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 124–133, New York, NY, USA, 2005. ACM. ISBN 1-59593-993-4. **Cited** on pages 6, 9, 43, 46, 57, 58, 114, 115, 133, 134 and 161.

Martin Erwig. Software engineering for spreadsheets. *IEEE Software*, 26(5):25–30, 2009. ISSN 0740-7459. **Cited** on page 35.

Martin Erwig and Margaret M. Burnett. Adding Apples and Oranges. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 173–191, London, UK, 2002. Springer-Verlag. ISBN 3-540-43092-X. **Cited** on page 6.

Martin Erwig, Robin Abraham, Irene Cooperstein, and Steve Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 136–145, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2. **Cited** on pages 6, 133 and 134.

EuSpRIG. European Spreadsheet Risks Interest Group. `http://www.eusprig.org/`, 2011. **Cited** on pages 3, 5 and 133.

Marc Fisher and Gregg Rothermel. The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms.

In *Proceedings of the First Workshop on End-user Software Engineering*, WEUSE I, pages 47–51, New York, NY, USA, 2005. ACM. ISBN 1-59593-131-7. **Cited** on page 85.

Marc Fisher, Mingming Cao, Gregg Rothermel, Curtis R. Cook, and Margaret M. Burnett. Automated test case generation for spreadsheets. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 141–153, New York, NY, USA, May 2002. ACM. ISBN 1-58113-472-X. **Cited** on page 6.

Gnumeric. The Gnome Office Spreadsheet. `http://projects.gnome.org/gnumeric/`, 2011. **Cited** on page 156.

Google. Google docs. `http://docs.google.com/`, 2011. **Cited** on page 3.

Graphviz. The Graphviz Tool. `http://www.graphviz.org/`, 2011. **Cited** on page 160.

Felienne Hermans, Martin Pinzger, and Arie van Deursen. Automatically extracting class diagrams from spreadsheets. In *ECOOP '10: Proceedings of the 24th European Conference on Object-Oriented Programming*, pages 52–75, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. **Cited** on page 6.

Ralf Hinze, Andres Löh, and Bruno Oliveira. "Scrap your boilerplate" reloaded. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer Berlin / Heidelberg, 2006. **Cited** on page 99.

Steven Holzner. *Eclipse Cookbook*. O'Reilly Media, Inc., May 2004. ISBN 0596007108. **Cited** on page 71.

Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27:1–52, May 1992. ISSN 0362-1340. **Cited** on page 151.

Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal*, 42(2):100–111, 1999. **Cited** on page 28.

Dorota Huizinga and Adam Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007. ISBN 0470042125, 9780470042120. **Cited** on page 5.

Tomas Isakowitz, Shimon Schocken, and Henry C. Lucas Jr. Toward a logical/physical theory of spreadsheet modelling. *ACM Transactions on Information Systems*, 13(1): 1–37, January 1995. **Cited** on page 6.

Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003. ISBN 0521826144. **Cited** on page 151.

Simon P. Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: A technical overview. In *Proceedings UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1993. **Cited** on page 30.

Matthijs F. Kuiper and João Saraiva. Lrc - A generator for incremental language-oriented tools. In *Proceedings of the 7th International Conference on Compiler Construction*, pages 298–301, London, UK, April 1998. Springer-Verlag. ISBN 3-540-64304-4. **Cited** on page 71.

Ralf Lämmel and Wolfgang Lohmann. Format Evolution. In *Proceedings of the 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001. **Cited** on page 12.

Ralf Lämmel and Joost Visser. A Strafunski application letter. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL ’03, pages 357–375, London, UK, UK, 2003. Springer-Verlag. ISBN 3-540-00389-4. **Cited** on pages 96 and 111.

Victoria Lemieux. *Competitive Viability, Accountability and Record Keeping: A Theoretical and Empirical Exploration Using a Case Study of Jamaican Commercial Bank Failures*. PhD thesis, University College London, 2002. **Cited** on page 6.

Victoria Lemieux. Archiving: The overlooked spreadsheet risk. *CoRR*, abs/0803.3231, 2008. **Cited** on page 6.

Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *EDBT '00: Proceedings of the 7th International Conference on Extending Database Technology*, pages 350–364, London, UK, 2000. Springer-Verlag. ISBN 3-540-67227-3. **Cited** on page 28.

David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. ISBN 0-914894-42-0. **Cited** on pages 22, 24, 25, 37 and 38.

Bertrand Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0-13-629155-4. **Cited** on pages 3 and 64.

Cupertino M. Miranda. Spreadsheets in Haskell. Graduation final assigment, Departamento de Informática, Universidade do Minho, 2004. **Cited** on page 156.

Roland Mittermeir and Markus Clermont. Finding high-level structures in spreadsheet programs. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 221–232, Washington, DC, USA, 2002. IEEE Computer Society. **Cited** on page 6.

Carroll Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, January 1990. ISSN 0001-5903. **Cited** on pages 12, 92, 96 and 118.

Robert J. Muller. *Database design for smarties: using UML for data modeling*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-515-0. **Cited** on page 56.

Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0262140535. **Cited** on page 1.

Noel Novelli and Rosine Cicchetti. FUN: An efficient algorithm for mining functional and embedded dependencies. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 189–203, London, UK, 2001. Springer-Verlag. ISBN 3-540-41456-8. **Cited** on pages 28 and 29.

Thomas M. O'Donovan. *VisiCalc Made Simple*. John Wiley & Sons, Inc., New York, NY, USA, 1984. ISBN 0471904570. **Cited** on page 2.

Linda O'Leary. *Microsoft Office Excel 2007 Introduction*. McGraw-Hill, Inc., New York, NY, USA, 2008. ISBN 0073294527, 9780073294520. **Cited** on pages 2 and 78.

José N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, 1990. **Cited** on pages 12, 92, 96 and 118.

José N. Oliveira. "Fractal" Types: An Attempt to Generalize Hash Table Calculation. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998. **Cited** on pages 97, 98 and 99.

José N. Oliveira. Functional dependency theory made 'simpler'. Technical Report PURe-05.01.01, DI-Research, January 2005. **Cited** on page 103.

José N. Oliveira. Transforming data by calculation. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE 2007*, volume 5235 of *Lecture Notes in Computer Science*, pages 134–195. Springer, 2008. ISBN 978-3-540-88642-6. **Cited** on pages 12, 92, 96 and 118.

OOoAuthors. *OpenOffice.org 3 Calc Guide*. Friends of OpenDocument Inc., 2010a. ISBN 978-1-921320-08-8. **Cited** on pages 78 and 80.

OOoAuthors. *Getting Started with OpenOffice.org 3*. CreateSpace, 2010b. ISBN 978-1440451775. **Cited** on page 78.

Oxford. Oxford dictionaries. `http://oxforddictionaries.com`, 2011. **Cited** on page 3.

Simon P. Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: Type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, July 2004. **Cited** on page 99.

Raymond R. Panko. Applying code inspection to spreadsheet testing. *Journal of Management Information Systems*, 16(2):159–176, 1999. ISSN 0742-1222. **Cited** on page 6.

Raymond R. Panko. Spreadsheet errors: What we know. What we think we can do. *Proceedings of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (EuSpRIG)*, July 2000. **Cited** on pages 3, 75, 82 and 133.

Raymond R. Panko and Salvatore Aurigemma. Revising the Panko-Halverson taxonomy of spreadsheet errors. *Decision Support Systems*, 49(2):235–244, 2010. ISSN 0167-9236. **Cited** on page 5.

Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: A roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 345–355, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. **Cited** on pages 135, 147 and 149.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1. **Cited** on page 4.

Andrew Pitonyak. *OpenOffice.org macros explained*. Hentzenwerke Publishing Inc., 2004. ISBN 978-1930919518. **Cited** on page 78.

Stephen G. Powell and Kenneth R. Baker. *The Art of Modeling with Spreadsheets*. John Wiley & Sons, Inc., New York, NY, USA, 2003. ISBN 0471209376. **Cited** on pages 3, 6, 45, 67, 133 and 138.

Kamalasen Rajalingham, David Chadwick, Brian Knight, and Dilwyn Edwards. Quality control in spreadsheets: A software engineering-based approach to spreadsheet development. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 4*, page 4006, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0493-0. **Cited** on page 6.

Kamalasen Rajalingham, David Chadwick, and Brian Knight. Classification of spreadsheet errors. *European Spreadsheet Risks Interest Group (EuSpRIG)*, 2001. **Cited** on pages 3 and 133.

Sudha Ram. Deriving functional dependencies from the entity-relationship model. *Communications of the ACM*, 38(9):95–111, 1995. ISSN 0001-0782. **Cited** on page 50.

Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 42–48, New York, NY, USA, 1984. ACM. ISBN 0-89791-131-8. **Cited** on page 71.

Jeffery A. Riley. *Introduction to OpenOffice.org*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2009. ISBN 0135073979, 9780135073971. **Cited** on page 2.

Steven A. Roman. *Writing Excel Macros with VBA*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2002. ISBN 0596003595. **Cited** on page 78.

Boaz Ronen, Michael Palley, and Henry Lucas Jr. Spreadsheet analysis and design. *Communications of the ACM*, 32(1):84–93, January 1989. **Cited** on page 6.

Gregg Rothermel, Margaret Burnett, Lixin Li, and Andrei Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10:110–147, 2001. **Cited** on page 6.

James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628. **Cited** on pages 9 and 63.

Jorma Sajaniemi. Modeling spreadsheet audit: A rigorous approach to automatic visualization. *Journal of Visual Languages and Computing*, 11:49–82, 2000. **Cited** on page 6.

Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 207–214, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2443-5. **Cited** on page 1.

Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN 0470025700. **Cited** on page 12.

J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988. ISBN 0-7167-8158-1. **Cited** on page 22.

UMHL. UMinho Haskell Software. `http://haskell.di.uminho.pt/UMHS`. **Cited** on page 152.

Mark G. J. van den Brand, Paul Klint, and Pieter A. Olivier. Compilation and memory management for ASF+SDF. In *Proceedings of the 8th International Conference* *on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 198–213, London, UK, 1999. Springer-Verlag. ISBN 3-540-65717-7. **Cited** on page 71.

Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40:831–873, July 2005. ISSN 0747-7171. **Cited** on pages 96 and 111.

Joost Visser. Coupled transformation of schemas, documents, queries, and constraints. *Electronic Notes on Theoretical Computer Science*, 200:3–23, May 2008. ISSN 1571-0661. **Cited** on pages 16, 92, 96 and 118.

Joost Visser and João Saraiva. Tutorial on strategic programming across programming paradigms. In *8th Brazilian Symposium on Programming Languages*, Niteroi, Brazil, May 2004. **Cited** on pages 96 and 111.

XML. Extensible Markup Language (XML) 1.0 (Fifth Edition). `http://www.w3.org/TR/REC-xml/`, 2008. **Cited** on page 13.

Hong Yao and Howard J. Hamilton. Mining functional dependencies from data. In M. J. Zaki, editor, *Data Mining and Knowledge Discovery*. Springer Netherlands, 2007. **Cited** on page 28.

Alan Yoder and David Cohn. Real spreadsheets for real programmers. In Henri E. Bal, editor, *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages*, pages 20–30. IEEE Computer Society, May 1994. **Cited** on page 6.

Index