Model-based Programming Environments for Spreadsheets*

Jácome Cunha[1][3], João Saraiva[1], and Joost Visser[2]

[1] HASLab / INESC TEC, Universidade do Minho, Portugal
{jacome,jas}@di.uminho.pt
[2] Software Improvement Group & Radboud University Nijmegen, The Netherlands
j.visser@sig.eu
[3] Escola Superior de Tecnologia e Gestão de Felgueiras, IPP, Portugal

Abstract. Although spreadsheets can be seen as a flexible programming environment, they lack some of the concepts of regular programming languages, such as structured data types. This can lead the user to edit the spreadsheet in a wrong way and perhaps cause corrupt or redundant data.

We devised a method for extraction of a relational model from a spreadsheet and the subsequent embedding of the model back into the spreadsheet to create a model-based spreadsheet programming environment. The extraction algorithm is specific for spreadsheets since it considers particularities such as layout and column arrangement. The extracted model is used to generate formulas and visual elements that are then embedded in the spreadsheet helping the user to edit data in a correct way.

We present preliminary experimental results from applying our approach to a sample of spreadsheets from the EUSES Spreadsheet Corpus.

1 Introduction

Developments in programming languages are changing the way in which we construct programs: naive text editors are now replaced by powerful programming language environments which are specialized for the programming language under consideration and which help the user throughout the editing process. Helpful features like highlighting keywords of the language or maintaining a beautified indentation of the program being edited are now provided by several text editors. Recent advances in programing languages extend such naive editors to powerful language-based environments [1–6]. Language-based environments use *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

* The authors would like to thank Martin Erwig and his team for providing us the code from the UCheck project. This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010048. The first author is supported by the FCT grant SFRH/BPD/73358/2010.

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 and more reliable programs.

Spreadsheet systems can be viewed as programming environments for non-professional programmers. These so-called *end-user* programmers vastly outnumber professional programmers [7].

In this paper, we propose a technique to enhance a spreadsheet system with mechanisms to guide end users to introduce correct data. A background process adds formulas and visual objects to an existing spreadsheet, based on a relational database schema. To obtain this schema, or model, 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 in introducing correct data. The knowledge about the spreadsheet under consideration is based on the *meaning* of its data that we infer using data mining and database normalization techniques.

Data mining techniques specific to spreadsheets are used to infer *functional dependencies* from the spreadsheet data. These functional dependencies define how certain spreadsheet columns determine the values of other columns. Database normalization techniques, namely the use of normal forms [8], are used to eliminate redundant functional dependencies, and to define a relational database model. Knowing the relational database model induced by the spreadsheet 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 three types of advanced features: *auto-completion of column values*, *non-editable columns* and *safe deletion of rows*.

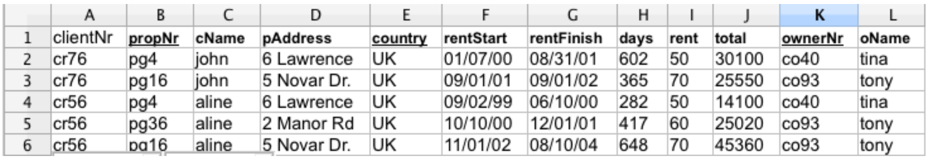
Our techniques work not only for database-like spreadsheets, like the example we will use throughout the paper, but they work also for realistic spreadsheets defined in other contexts (for example, inventory, grades or modeling). In this paper we present our first experimental results obtained by considering a large set of spreadsheets included in the EUSES Spreadsheet Corpus [9].

This paper is organized as follows. Section 2 presents an example used throughout the paper. Section 3 presents our algorithm to infer functional dependencies and how to construct a relational model. Section 4 discusses how to embed assisted editing features into spreadsheets. A preliminary evaluation of our techniques is present in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

2 A Spreadsheet Programming Environment

In order to present our approach we shall consider the following well-known example taken from [10] and modeled in a spreadsheet as shown in Figure 1.

This spreadsheet contains information related to a housing rental system. It gathers information about clients, owners, properties, prices and rental periods. The name of each column gives a clear idea of the information it represents. We extend this example



	A	B	C	D	E	F	G	H	I	J	K	L
1	clientNr	**propNr**	**cName**	**pAddress**	**country**	**rentStart**	**rentFinish**	**days**	**rent**	**total**	**ownerNr**	**oName**
2	cr76	pg4	john	6 Lawrence	UK	01/07/00	08/31/01	602	50	30100	co40	tina
3	cr76	pg16	john	5 Novar Dr.	UK	09/01/01	09/01/02	365	70	25550	co93	tony
4	cr56	pg4	aline	6 Lawrence	UK	09/02/99	06/10/00	282	50	14100	co40	tina
5	cr56	pg36	aline	2 Manor Rd	UK	10/10/00	12/01/01	417	60	25020	co93	tony
6	cr56	pg16	aline	5 Novar Dr.	UK	11/01/02	08/10/04	648	70	45360	co93	tony

Fig. 1. A spreadsheet representing a property rental system.

with three additional columns, named $days$ (that computes the total number of rental days by subtracting the column $rentStart$ to $rentFinish$), $total$ (that multiplies the number of rental days by the rent per day value, $rent$) and $country$ (that represents the property's country). As usually in spreadsheets, the columns $days$ and $rent$ are expressed by formulas.

This spreadsheet defines a valid model to represent the information of the rental system. However, it contains redundant information: the displayed data specifies the house rental of two clients (and owners) only, but their names are included five times, for example. 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, thus corrupting the data on the spreadsheet.

Two common problems occur as a consequence of redundant data: *update anomalies* and *deletion anomalies* [11]. The former problem occurs when we change information in one place but leave the same information unchanged in the other places. The problem also occurs if the update is not performed exactly in the same way. In our example, this happens if we change the rent of property number `pg4` from `50` to `60` only one row and leave the others unchanged, for example. The latter problem occurs when we delete some data and lose other information as a side effect. For example, if we delete row 5 in the our example all the information concerning property `pg36` is lost.

The database community has developed techniques, such as data normalization, to eliminate such redundancy and improve data integrity [11, 12]. Database normalization is based on the detection and exploitation of functional dependencies inherent in the data [13]. Can we leverage these database techniques for spreadsheets systems so that the system eliminates the update and deletion anomalies by guiding the end user to introduce correct data? Based on the data contained in our example spreadsheet, we would like to discover the following functional dependencies which represent the four entities involved in our house rental system: *countries*, *clients*, *owners* and *properties*.

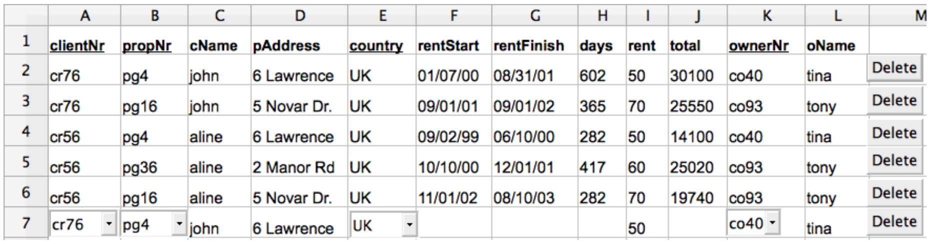
$$\begin{array}{ll}
country & \rightharpoonup \\
clientNr & \rightharpoonup cName \\
ownerNr & \rightharpoonup oName \\
propNr & \rightharpoonup pAddress, rent, ownerNr
\end{array}$$

A functional dependency $A \rightharpoonup B$ means that if we have two equal inhabitants of A, then the corresponding inhabitants of B are also equal. For instance, the client number functionally determines his/her name, since no two clients have the same number. The right hand side of a functional dependency can be an empty set. This occurs, for example, in the $country$ functional dependency. Note that there are several columns (labeled $rentStart$, $rentFinish$, $days$ and $total$) that are not included in any functional dependency. This happens because their data do not define any functional dependency.

Using these functional dependencies it is possible to construct a relational database schema. Each functional dependency is translated into a table where the attributes are the ones participating in the functional dependency and the primary key is the left hand side of the functional dependency. In some cases, foreign keys can be inferred from the schema. The relational database schema can be normalized in order to eliminate data redundancy. A possible normalized relational database schema created for the house rental spreadsheet is presented bellow.

$\underline{country}$
$\underline{clientNr}, cName$
$\underline{ownerNr}, oName$
$\underline{propNr}, pAddress, rent, ownerNr$

This database schema defines a table for each of the entities described before. Having defined a relational database schema we would like to construct a spreadsheet environment that respects that relational model, as shown in Figure 2.

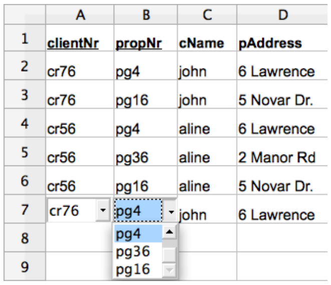


	A	B	C	D	E	F	G	H	I	J	K	L	M
1	clientNr	propNr	cName	pAddress	country	rentStart	rentFinish	days	rent	total	ownerNr	oName	
2	cr76	pg4	john	6 Lawrence	UK	01/07/00	08/31/01	602	50	30100	co40	tina	Delete
3	cr76	pg16	john	5 Novar Dr.	UK	09/01/01	09/01/02	365	70	25550	co93	tony	Delete
4	cr56	pg4	aline	6 Lawrence	UK	09/02/99	06/10/00	282	50	14100	co40	tina	Delete
5	cr56	pg36	aline	2 Manor Rd	UK	10/10/00	12/01/01	417	60	25020	co93	tony	Delete
6	cr56	pg16	aline	5 Novar Dr.	UK	11/01/02	08/10/03	282	70	19740	co93	tony	Delete
7	cr76	pg4	john	6 Lawrence	UK				50		co40	tina	Delete

Fig. 2. A spreadsheet with auto-completion based on relational tables.

For example, this spreadsheet would not allow the user to introduce two different properties with the same property number *propNr*. 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. Figure 3 shows a possible spreadsheet environment where possible properties can be chosen from a *combo box*.

Using the relational data base schema we would like that our spreadsheet offers the following features:



	A	B	C	D
1	clientNr	propNr	cName	pAddress
2	cr76	pg4	john	6 Lawrence
3	cr76	pg16	john	5 Novar Dr.
4	cr56	pg4	aline	6 Lawrence
5	cr56	pg36	aline	2 Manor Rd
6	cr56	pg16	aline	5 Novar Dr.
7	cr76	pg4	john	6 Lawrence
8				
9				

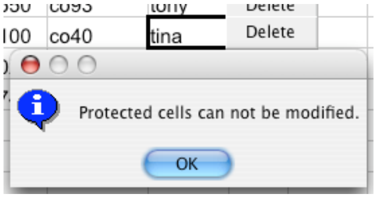
Fig. 3. Selecting possible values of columns using a combo box.

Auto-completion of Column Values: The columns corresponding to primary keys in the relational model determine the values of other columns; we want the spreadsheet environment to be able to automatically fill those columns provided the end -user defines the value of the primary key.

For example, the value of the property number (*propNr*, column `B`) determines the values of the address (*pAddress*, column `D`), rent per day (*rent*, column `I`), and owner number (*ownerNr*, column `K`). Consequently, the spreadsheet environment should be able to automatically fill in the values of the columns `D`, `I` and `K`, given the value of column `B`. Since *ownerNr* (column `K`) is a primary key of another table, transitively the value of *oName* (column `L`) is also defined. This auto-completion mechanism has been implemented and is presented in the spreadsheet environment of Figure 2.

Non-Editable Columns: Columns that are part of a table but not part of its primary key must not be editable. For example, column `L` is part of the owner table but it is not part of its primary key. Thus, it must be protected from being edited. The primary key of a table must not be editable also since it can destroy the dependency. This feature prevents the end user from introducing potentially incorrect data and, thus, producing update anomalies. Figure 4 illustrates this edit restriction.

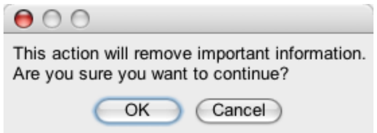
Safe Deletion of Rows: Another usual problem with non-normalized data is the deletion problem. Suppose in our running example that row 5 is deleted. In such scenario, all the information about the `pg36` property is lost. However, it is likely that the user wanted to delete the rental transaction represented by that row only. In order to prevent this type of deletion problems, we have added a button per spreadsheet row (see Figure 2). When pressed, this button detects whether the end user is deleting important information included in the corresponding row. In case important information is removed by such deletion, a warning window is displayed, as shown in Figure 5.



Protected cells can not be modified.

Fig. 4. In order to prevent update anomalies some columns must not be editable.

Apart from these new features, the user can still access traditional editing features, and can rely on recalculation of functional dependencies in the background.



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

Fig. 5. Window to warn the end user that crucial information may be deleted.

Traditional Editing: Advanced programming language environments provide both advanced editing mechanisms and traditional ones (*i.e.*, 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 relational database model that the spreadsheet data induces.

Recalculation of the Relational Database Model: Because standard editing allows the end user to introduce data violating the underlying relational model, we would like that the spreadsheet environment may 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 relational model. However, when the end user returns to advanced editing, then the spreadsheet should infer a new relational model that will be used in future (advanced) interactions.

In this section we have described an instance of our techniques. In fact, the spreadsheet programming environment shown in the Figures 2, 3, 4 and 5 was automatically produced from the original spreadsheet displayed in Figure 1. In the following sections we will present in detail the technique to perform such an automatic spreadsheet refactoring.

3 From Spreadsheets to Relational Databases

This section briefly explains how to extract functional dependencies from the spreadsheet data and how to construct a normalized relational database schema modeling such data. These techniques were introduced in detail in our work on defining a bidirectional mapping between spreadsheets and relational databases [14]. In this section we briefly present an extension to that algorithm that uses spreadsheet specific properties in order to infer a more realistic set of functional dependencies.

Relational Databases: 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. A *relation* (or *table*) r on a relation 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. A *relational database schema* is a collection of relation 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 a *Primary Key* (PK). On certain occasions there may be more then one set suitable for becoming the primary key. They are designated *candidate keys* and only one is chosen to become primary key. A *Foreign Key* (FK) is a set of attributes within one relation that matches the primary key of some relation.

The normalization of a database is important to prevent data redundancy. Although there are several different normal forms, in general, a RDB is considered normalized if it respects the *Third Normal Form* (3NF) [10].

Discovering Functional Dependencies: In order to define the RDB schema, we first need to compute the functional dependencies presented in a given spreadsheet data. In [14] we reused the well known data mining algorithm, named FUN, to infer such dependencies. This algorithm was developed in the context of databases with the main goal of inferring all existing functional dependencies in the input data. As a result, FUN may infer a large set of functional dependencies depending on the input data. For our example, we list the functional dependencies inferred from the data using FUN:

$$\begin{array}{ll}
\mathit{clientNr} & \rightharpoonup \mathit{cName}, \mathit{country} \\
\mathit{propNr} & \rightharpoonup \mathit{country}, \mathit{pAddress}, \mathit{rent}, \mathit{ownerNr}, \mathit{oName} \\
\mathit{cName} & \rightharpoonup \mathit{clientNr}, \mathit{country} \\
\mathit{pAddress} & \rightharpoonup \mathit{propNr}, \mathit{country}, \mathit{rent}, \mathit{ownerNr}, \mathit{oName} \\
\mathit{rent} & \rightharpoonup \mathit{propNr}, \mathit{country}, \mathit{pAddress}, \mathit{ownerNr}, \mathit{oName} \\
\mathit{ownerNr} & \rightharpoonup \mathit{country}, \mathit{oName} \\
\mathit{oName} & \rightharpoonup \mathit{country}, \mathit{ownerNr}
\end{array}$$

Note that, data contained in the spreadsheet exhibits all those dependencies. In fact, even the non-natural dependency $\mathit{rent} \rightharpoonup \mathit{propNr}, \mathit{country}, \mathit{pAddress}, \mathit{ownerNr}, \mathit{oName}$ is inferred. Indeed, the functional dependencies derived by the FUN algorithm depend heavily on the quantity and quality of the data. Thus, for small samples of data, or data that exhibits too many or too few dependencies, the FUN algorithm may not produce the desired functional dependencies.

Note also that the $\mathit{country}$ column occurs in most of the functional dependencies although only a single country actually appears in a column of the spreadsheet, namely UK. Such single value columns are common in spreadsheets. However, for the FUN algorithm they induce redundant fields and redundant functional dependencies.

In order to derive more realistic functional dependencies for spreadsheets we have extended the FUN algorithm so that it considers the following spreadsheet properties:

- *Single value columns*: these columns produce a single functional dependency with no right hand side ($\mathit{country} \rightharpoonup$, for example). This columns are not considered when finding other functional dependencies.
- *Semantic of labels*: we consider label names as strings and we look for the occurrence of words like *code, number, nr, id* given them more priority when considered as primary keys.
- *Column arrangement*: we give more priority to functional dependencies that respect the order of columns. For example, $\mathit{clientNr} \rightharpoonup \mathit{cName}$ has more priority than $\mathit{cName} \rightharpoonup \mathit{clientNr}$.

Moreover, to minimize the number of functional dependencies we consider the smallest subset that includes all attributes/columns in the original set computed by FUN. The result of our spreadsheet functional dependency inference algorithm is:

$$\begin{array}{ll}
\mathit{country} & \rightharpoonup \\
\mathit{clientNr} & \rightharpoonup \mathit{cName} \\
\mathit{ownerNr} & \rightharpoonup \mathit{oName} \\
\mathit{propNr} & \rightharpoonup \mathit{pAddress}, \mathit{rent}, \mathit{ownerNr}, \mathit{oName}
\end{array}$$

This set of dependencies is very similar to the one presented in the previous section. The exception is the last functional dependency which has an extra attribute (oName).

Spreadsheet Formulas: Spreadsheets use formulas to define the values of some elements in terms of other elements. For example, in the house rental spreadsheet, the column days is computed by subtracting the column *rentFinish* from *rentStart*, and it is usually written as follows `H3 = G3 - F3`. This formula states that the values of `G3` and `F3` determine the value of `H3`, thus inducing the following functional dependency: $rentStart, rentFinish \rightharpoonup days$.

Formulas can have references to other formulas. Consider, for example, the second formula of the running example `J3 = H3 * I3`, which defines the total rent by multiplying the total number of days by the value of the rent. Because `H3` is defined by another formula, the values that determine `H3` also determine `J3`. As a result, the two formulas induce the following functional dependencies:

$$\begin{array}{ll} \mathit{rentStart}, \mathit{rentFinish} & \rightharpoonup \mathit{days} \\ \mathit{rentStart}, \mathit{rentFinish}, \mathit{rent} & \rightharpoonup \mathit{total} \end{array}$$

In general, a spreadsheet formula of the following form $X_0 = f(X_1, \ldots, X_n)$ induces the following functional dependency: $X_1, \ldots, X_n \rightharpoonup X_0$. In spreadsheet systems, formulas are usually introduced by copying them through all the elements in a column, thus making the functional dependency explicit in all the elements. This may not always be the case and some elements can be defined otherwise (e.g. by using a constant value or a different formula). In both cases, all the cells referenced must be used in the antecedent of the functional dependency.

These functional dependencies are useful for the mapping of spreadsheets to databases as presented in [14]. In this work, they are not relevant since the existing formulas are used to fill in those columns.

Normalizing Functional Dependencies: Having computed the functional dependencies, we can now normalize them. Next, we show the results produced by the *synthesize* algorithm introduced by Maier in [15]. The *synthesize* algorithm receives a set of functional dependencies as argument and returns a new 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 subsets of a scheme R and Y is also a subset of R. A relation r satisfies the CFD $(X_1, \ldots, X_n) \rightharpoonup Y$ if it satisfies the functional dependencies $X_i \rightharpoonup X_j$ and $X_i \rightharpoonup Y$, where $1 \leqslant i$ and $j \leqslant k$. In a 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*.

Next, we list the compound functional dependencies computed from the functional dependencies induced by our running example.

$$\begin{array}{ll} (\{\mathit{country}\}) & \rightharpoonup \{\} \\ (\{\mathit{clientNr}\}) & \rightharpoonup \{\mathit{cName}\} \\ (\{\mathit{ownerNr}\}) & \rightharpoonup \{\mathit{oName}\} \\ (\{\mathit{propNr}\}) & \rightharpoonup \{\mathit{pAddress}, \mathit{rent}, \mathit{ownerNr}\} \end{array}$$

Computing the Relational Database Schema: Each compound functional dependency defines several candidate keys for each table. However, to fully characterize the relational database schema we need to choose the primary key from those candidates. To find such keys we use a simple algorithm: we produce all the possible tables using each candidate key as the primary key; we then use the same algorithm that is used to choose the initial functional dependencies to choose the best table. Note that before applying the *synthesize* algorithm, all the functional dependencies with antecedents' attributes representing formulas should be eliminated since a primary key must not change over time. The final result is listed bellow.

$\underline{country}$
$\underline{clientNr}, cName$
$\underline{ownerNr}, oName$
$\underline{propNr}, pAddress, rent, ownerNr$

This relational database model corresponds exactly to the one shown in Section 2. Note that the synthesize algorithm removed the redundant attribute $oName$ that occurred in the last functional dependency.

4 Building Spreadsheet Programming Environments

This section presents techniques to refactor spreadsheets into powerful spreadsheet programming environments as described in Section 2. This spreadsheet refactoring is implemented as the embedding of the inferred functional dependencies and the computed relational model in the spreadsheet. This embedding is modeled in the spreadsheet itself by standard formulas and visual objects: formulas are added to the spreadsheet to guide end users to introduce correct data.

Before we present how this embedding is defined, let us first define a spreadsheet. A spreadsheet can be seen as a partial function $S : A \to V$ mapping addresses to spreadsheet values. Elements of S are called *cells* and are represented as (a, v). A cell address is taken from the set $A = \mathbb{N} \times \mathbb{N}$. A value $v \in V$ can be an input plain value $c \in C$ like a string or a number, references to other cells using addresses or formulas $f \in F$ that can be applied to one or more values: $v \in V ::= c \mid a \mid f(v, \ldots, v)$.

Auto-completion of Column Values: This feature is implemented by embedding each of the relational tables in the spreadsheet. It is implemented by a spreadsheet formula and a combo box visual object. The combo box displays the possible values of one column, associated to the primary key of the table, while the formula is used to fill in the values of the columns that the primary key determines.

Let us consider the table $\underline{ownerNr}, oName$ from our running example. In the spreadsheet, $ownerNr$ is in column K and $oName$ in column L. This table is embed in the spreadsheet introducing a combo box containing the existing values in the column K (as displayed in Figure 2). Knowing the value in the column K we can automatically introduce the value in column L. To achieve this, we embed the following formula in row 7 of column L:

$$S\ (L, 7) = \textbf{if}\ (\textbf{isna}\ (\textbf{vlookup}\ (K7, K2 : L6, 2, 0)), \texttt{""}, \textbf{vlookup}\ (K7, K2 : L6, 2, 0))$$

This formula uses a (library) function **isna** to test if there is a value introduced in column K. In case that value exists, it searches (with the function **vlookup**) the corresponding value in the column L and references it. If there is no selected value, it produces the empty string. The combination of the combo box and this formula guides the user to introduce correct data as illustrated in Figure 2.

We have just presented a particular case of the formula and visual object induced by a relational table. Next we present the general case. 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 relational database table $\underline{a_1},...,\underline{a_n},c_1,...,c_m$, with $a_1,...,a_n,c_1,...,c_m$ column indexes of the spreadsheet, induces firstly, a combo box defined as follows:

$$
\begin{array}{l}
\forall\ c\ \in\ \{a_1,...,a_n\}, \forall\ r\ \in\ \{minr,...,maxr\}:\\
\quad S\ (c,r) = \mathit{combobox} := \{\,\mathit{linked\_cell}\ \ := (c,r);\\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{source\_cells} := (c,\mathit{r1}) : (c,r-1)\}
\end{array}
$$

secondly, a spreadsheet formula defined as:

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

This formula must be used for each non primary key column created by our algorithm. Each conditional **if** inside the main **if** is responsible for checking a primary key column. In the case a primary key column value is chosen, **isna** (**vlookup** (...)), the formula calculates the corresponding non primary key column value, **vlookup** (...). If the values chosen by all primary key columns are the same, then that value is used in the non primary key column. This formula considers tables with primary keys consisting of multiple attributes (columns). Note also that the formula is defined in each column associated to non-key attribute values.

The example table analysed before is an instance of this general one. In the table $\underline{\mathit{ownerNr}}, \mathit{oName}$, $\mathit{ownerNr}$ is a_1, oName is c_1, c is L, $\mathit{r1}$ is 2, minr is 7. The value of $maxr$ is always the last row supported by the spreadsheet system.

Foreign keys pointing to primary keys become very helpful in this setting. For example, if we have the relational tables $\underline{A}, B$ and $\underline{B}, C$ where B is a foreign key from the second table to the first one, then when we perform auto-completion in column A, both B and C are automatically filled in. This was the case presented in Figure 2.

Non-Editable Columns: To prevent wrong introduction of data, and thus, producing update anomalies, we protect some columns from edition. A relational table, such as $\underline{a_1},...,\underline{a_n},c_1,...,c_m$, induces the non-edition of columns $a_1,...,a_n,c_1,...,c_m$. That is to say that all columns that form a table become non-editable. Figure 4 illustrates such a restriction. In the case where the end user really needs to change the value of such protected columns, we provide traditional editing (explained below).

Safe Deletion of Rows: Another usual problem with non-normalized data is the deletion of data. Suppose in our running example that row 5 is deleted. All the information about property `pg36` is lost, although the user would probably want to delete that rental transaction only. To correctly delete rows in the spreadsheet, a button is added to each row in the spreadsheet as follows: for each relational table $\underline{a_1, ..., a_n}, c_1, ..., c_m$ each button checks, on its corresponding row, the columns that are part of the primary key, $a_1, ..., a_n$. For each primary key column, it verifies if the value to remove is the last one.

Let $c \in \{a_1, ..., a_n\}$, let 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 as follows:

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

If the value is the last one, the spreadsheet warns the user (**showMessage**) as can be seen in Figure 5. If the user presses the `OK` button, the spreadsheet will remove the row. In the other case, `Cancel`, no action will be performed. In the case the value is not the last one, the row will simply be removed, **deleteRow** (r). For example, in column *propNr* of our running example, the row 5 contains the last data about the house with code `pg36`. If the user tries to delete this row, the warning will be triggered.

Traditional Editing Advanced programming language environments provide both advanced editing mechanisms and traditional ones (*i.e.*, text editing). In a similar way, a spreadsheet environment should allow the user to perform traditional spreadsheet editing too. Thus, the environment should provide a mechanism to enable/disable the advanced features described in this section. When advanced features are disabled, the end user is be able to introduce data that violates the (previously) inferred relational model. However, when the end user returns to advance editing, the spreadsheet infers a new relational model that will be used in future (advanced) interactions.

4.1 HaExcel Add-in

We have implemented the FUN algorithm, the extensions described in this paper, the *synthesize* algorithm, and the embedding of the relational model in the HASKELL programming language [16]. We have also defined the mapping from spreadsheet to relational databases in the same framework named HaExcel [14]. Finally, we have extended this framework to produce the visual objects and formulas to model the relational tables in the spreadsheet. An Excel add-in as been also constructed so that the end user can use spreadsheets in this popular system and at the same time our advanced features.

5 Preliminary Experimental Results

In order to evaluate the applicability of our approach, we have performed a preliminary experiment on the EUSES Corpus [9]. This corpus was conceived as a shared resource to support research on technologies for improving the dependability of spreadsheet programming. It contains more than 4500 spreadsheets gathered from different sources and developed for different domains. These spreadsheets are assigned to eleven different categories. including financial (containing 19% of the total number of spreadsheets), inventory (17%), homework (14%), grades (15%), database (17%) and modeling (17%) (the remaining 1% represents other spreadsheets). Among the spreadsheets in the corpus, about 4.4% contain macros, about 2.3% contain charts, and about 56% do not have formulas being only used to store data.

In our preliminary 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 1): a few spreadsheets failed to parse, due to glitches in the Excel to Gnumeric conversion (which we use to bring spreadsheets into a processable form). Other spreadsheets were parsed, but no tables could be recognized in them, *i.e.*, their users did not adhere to any of the supported layout conventions. The layout conventions we support are the ones presented in the UCheck project [17]. This was the case for about one third of the spreadsheets in our item. The other spreadsheets were parsed, tables were recognized, and edit assistance was generated for them. We will focus on the last groups in the upcoming sections.

Processed Spreadsheets: The results of processing our sample of spreadsheets from the EUSES corpus are summarized in Table 1. The rows of the table are grouped by category as documented in the corpus. The first three columns contain size metrics on the spreadsheets. They indicate how many tables were recognized, how many columns are present in these tables, and how many cells. For example, the first spreadsheet in the *financial* category contains 15 tables with a total of 65 columns and 242 cells.

File name	Recognized tables	Cols.	Cells	FDs	Cols. w/ safe insertion & deletion	Auto-compl. cols.	Non-editab. cols.
cs101							
Act4_023_capen	5	24	402	0	0	0	0
act3_23_bartholomew	6	21	84	1	8	1	9
act4_023_bartholomew	6	23	365	0	0	0	0
meyer_Q1	2	8	74	0	0	0	0
posey_Q1	5	23	72	0	8	0	8
database							
%5CDepartmental%20Fol#A8...	2	4	3463	0	0	0	0
00061r0P802-15_TG2-Un...69	23	55	491	0	18	4	21
00061r5P802-15_TG2-Un...6C	30	83	600	25	21	5	26
0104TexasNutrientdb	5	7	77	1	1	1	2
01BTS_framework	52	80	305	4	23	2	25
03-1-report-annex-5	20	150	1599	12	15	8	22
filby							
BROWN	5	14	9047	2	3	1	4
CHOFAS	6	48	4288	3	3	1	4
financial							

continues on the next page

Table 1 – continuation of previous page

File name	Recognized tables	Cols.	Cells	FDs	Cols. w/ safe insertion & deletion	Auto-compl. cols.	Non-editab. cols.
03PFMJOURnalBOOKSFina...	15	65	242	0	7	0	7
10-formc	12	20	53	8	5	4	9
forms3							
ELECLAB3.reichwja.xl97	1	4	44	0	0	0	0
burnett-clockAsPieChart	3	8	14	0	1	0	1
chen-heapSortTimes	1	2	24	0	0	0	0
chen-insertSortTimes	1	2	22	0	0	0	0
chen-lcsTimes	1	2	22	0	0	0	0
chen-quickSortTimes	1	2	24	0	0	0	0
cs515_npeg_chart.reichwja.xl97	7	9	93	0	0	0	0
cs515_polynomials.reichwja.xl97	6	12	105	0	0	0	0
cs515_runtimeData.reichwja.X...	2	6	45	0	0	0	0
grades							
0304deptcal	11	41	383	19	18	17	28
03_04ballots1	4	20	96	6	4	0	4
030902	5	20	110	0	0	0	0
031001	5	20	110	0	0	0	0
031501	5	15	51	31	3	1	4
homework							
01_Intro_Chapter_Home#A9171	6	15	2115	0	1	0	1
01readsdis	4	16	953	5	4	3	6
02%20fbb%20medshor	1	7	51	0	0	0	0
022timeline4dev	28	28	28	0	0	0	0
026timeline4dev	28	28	30	0	2	0	2
03_Stochastic_Systems#A9172	4	6	48	0	2	0	2
04-05_proviso_list	79	232	2992	0	25	0	25
inventory							
02MDE_framework	50	83	207	10	31	1	32
02f202assignment%234soln	37	72	246	7	20	1	21
03-1-report-annex-2	5	31	111	10	5	5	8
03singapore_elec_gene#A8236	9	45	153	3	5	2	7
0038	10	22	370	0	0	0	0
modeling							
%7B94402d63-cdd8-4cc3#A…	1	3	561	0	0	0	0
%EC%86%90%ED%97%8C…	1	10	270	13	7	5	9
%EC%9D%98%EB%8C%80…	1	7	1442	4	4	5	6
%EC%A1%B0%EC%9B%90…	2	17	534	18	13	5	15
%ED%99%98%EA%B2%B…	3	7	289	2	1	2	3
0,10900,0-0-45-109057-0,00	4	14	6558	9	9	2	10
00-323r2	24	55	269	31	9	6	15
00000r6xP802-15_Docum#A…	3	13	3528	10	9	3	11
003_4	25	50	2090	0	0	0	0

Table 1: Preliminary results of processing the selected spreadsheets.

The fourth column shows how many functional dependencies were extracted from the recognized tables. These are the non-trivial functional dependencies that remain after we use our extension to the FUN algorithm to discard redundant dependencies. The last three 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) functional dependencies are extracted from the recognized tables. In the other cases, the three columns respectively indicate:

- For how many columns a combo box has been generated for *controlled insertion*. The same columns are also enhanced with the *safe deletion of rows* feature.
- For how many columns the *auto-completion of column values* has been activated, *i.e.*, for how many columns the user is no longer required to insert values manually.
- How many columns are locked to prevent edit actions where information that does not appear elsewhere is deleted inadvertently.

For example, for the first spreadsheet of the *inventory* category, combo boxes have been generated for 31 columns, auto-completion has been activated for 1 column, and locking has been applied to 32 columns. Note 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 (more than 150,000 cells).

Observations: On the basis of these preliminary results, a number of interesting observations can be made. For some categories, edit assistance is successfully added to almost all spreadsheets (e.g. *inventory* and *database*), while for others almost none of the spreadsheets lead to results (e.g. the *forms/3* category). The latter may be due to the small sizes of the spreadsheets in this category. For the *financials* category, we can observe that in only 2 out of 10 sample spreadsheets tables were recognized, but edit assistance was successfully generated for both of these.

The *percentage* of columns for which edit assistance was generated varies. The highest percentage was obtained for the second spreadsheet of the *modeling* category, with 9 out of 10 columns (90 %). A good result is also obtained for the first spreadsheet of the *grades* category with 28 out of 41 columns (68.3 %). On the other hand, the 5th of the *homework* category gets edit assistance for only 2 out of 28 columns (7.1 %). The number of columns with combo boxes often outnumbers the columns with auto-completion. This may be due to the fact that many of the functional dependencies are small, with many having only one column in the antecedent and none in consequent.

Evaluation: Our preliminary experiment justifies two preliminary conclusions. Firstly, the tool is able to successfully add edit assistance to a series of non-trivial spreadsheets. A more thorough study of these and other cases can now be started to identify technical improvements that can be made to the algorithms for table recognition and functional dependency extraction. 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.

6 Related Work

Our work is strongly related to a series of techniques by Abraham *et al.*. Firstly, they designed and implemented an algorithm that uses the labels within a spreadsheet for *unit checking* [18, 19]. By typing the cells in a spreadsheet with unit information and tracking them through references and formulas, various types of users errors can be caught. We have adopted the view of Abraham *et. al.*of a spreadsheet as a collection of tables and we have reused their algorithm for identifying the spatial boundaries of these tables. Rather than exploiting the labels in the spreadsheet to reconstruct implicit user intentions, we exploit redundancies in data elements. Consequently, the errors caught by our approach are of a different kind. Secondly, Abraham *et. al.*developed a type system and corresponding inference algorithm that assigns types to values, operations, cells, formulas, and entire spreadsheets [20]. The type system can be used to catch errors in spreadsheets or to infer spreadsheet models that can help to prevent future errors. We have used such spreadsheet models, namely the ClassSheet models [21], to realize model-driven software evolution in the context of spreadsheets [22–28].

In previous work we presented techniques and tools to transform spreadsheets into relational databases and back [14]. We used the FUN algorithm to construct a relational model, but rather than generating edit assistance, the recovered information was used to perform spreadsheet *refactoring*. The algorithm for extracting and filtering spreadsheets presented in the current paper is an improvement over the algorithm that we used previously.

We provided a short user-centered overview of the idea of generating edit assistance for spreadsheets via extraction of functional dependencies in a previous short paper [29]. In the current paper, we have provided the technical details of the solution, including the improved algorithm for extraction and filtering functional dependencies. Also, we have provided the first preliminary evaluation of the approach by application to a sample of spreadsheets from the EUSES corpus.

7 Conclusions

Contributions: 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 improved approach for mining functional dependencies from spreadsheets and subsequent synthesis of a relational database. On this basis, we have made the following contributions:

- Derivation of formulas and visual elements that capture the knowledge encoded in the reconstructed relational database schema.
- 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 reconstruction of a schema, for derivation of corresponding formulas and visual elements, and for their embedding into a *add-in* for spreadsheet environments.

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

Future Work: There are several extensions of our work that we would like to explore. The algorithms running in the background need to recalculate the relational schema 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, for example, by use of incremental evaluation. Our approach could be integrated with similar, complementary approaches to cover a wider range of possible user errors. In particular, the work of Abraham *et al.* [20, 30] for preventing range, reference, and type errors could be combined with our work for preventing data loss and inconsistency. We have presented some preliminary experimental results to pave the way for a more comprehensive validation experiments. In particular, we intend to set up a structured experiment for testing the impact on end-user productivity, and effectiveness.

References

1. Saraiva, J.: Design, Implementation and Animation of Spreadsheets in the Lrc System. In: Int. workshop on Foundations of Spreadsheet, Martin Erwig editor, ENTCS (2004)
2. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In Koskimies, K., ed.: 7th International Conference on Compiler Construction. Volume 1383 of LNCS., Springer-Verlag (April 1998) 298–301
3. Reps, T., Teitelbaum, T.: The synthesizer generator. SIGSOFT Softw. Eng. Notes **9**(3) (1984) 42–48
4. van den Brand, M., Klint, P., Olivier, P.: Compilation and Memory Management for ASF+SDF. In Stefan Jähnichen, ed.: 8th International Conference on Compiler Construction. Volume 1575 of LNCS. (March 1999) 198–213
5. Saraiva, J., Swierstra, S.D.: Generating spreadsheet-like tools from strong attribute grammars. In Pfenning, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings. Volume 2830 of LNCS., Springer (2003) 307–323
6. Holzner, S.: Eclipse. O'Reilly (May 2004)
7. Scaffidi, C., Shaw, M., Myers, B.: Estimating the numbers of end users and end user programmers. VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (2005) 207–214
8. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6) (1970) 377–387
9. II, M.F., Rothermel, G.: The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In: Proceedings of the 1st Workshop on End-User Software Engineering. (2005) 47–51
10. Connolly, T., Begg, C.: Database Systems, A Practical Approach to Design, Implementation, and Management. Addison-Wesley, 3 edition (2002)
11. Ullman, J.D., Widom, J.: A First Course in Database Systems. Prentice Hall (1997)
12. Date, C.J.: An Introduction to Database Systems. Addison-Wesley (1995)
13. Beeri, C., Fagin, R., Howard, J.: A complete axiomatization for functional and multivalued dependencies in database relations. In: Proc. of the ACM SIGMOD Int. Conf. on Management of Data. (1977) 47–61

14. Cunha, J., Saraiva, J., Visser, J.: From spreadsheets to relational databases and back. In: PEPM'09: Proc. of the 2009 ACM SIGPLAN workshop on Partial Evaluation and Program manipulation, ACM (2009) 179–188
15. Maier, D.: The Theory of Relational Databases. Computer Science Press (1983)
16. Peyton Jones, S.: Haskell 98: Language and libraries. J. Funct. Program. **13**(1) (2003) 1–255
17. Abraham, R., Erwig, M.: UCheck: A spreadsheet type checker for end users. J. Vis. Lang. Comput. **18**(1) (2007) 71–95
18. Erwig, M., Burnett, M.: Adding apples and oranges. 4th Int. Symp. on Practical Aspects of Declarative Languages (2002) 173–191
19. Abraham, R., Erwig, M.: Header and unit inference for spreadsheets through spatial analyses. Visual Languages and Human Centric Computing, 2004 IEEE Symposium on (Sept. 2004) 165–172
20. Abraham, R., Erwig, M.: Type inference for spreadsheets. In Bossi, A., Maher, M.J., eds.: Proceedings of the 8th Int. ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy, ACM (2006) 73–84
21. Engels, G., Erwig, M.: ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In: ASE'05: Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering, ACM (2005) 124–133
22. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: VL/HCC'10: IEEE Symp. on Visual Languages and Human-Centric Computing, IEEE Computer Society (2010) 93–100
23. Beckwith, L., Cunha, J., ao Paulo Fernandes, J., Saraiva, J.: End-users productivity in model-based spreadsheets: An empirical study. In: Proceedings of the Third International Symposium on End-User Development. IS-EUD '11 (2011) 282–288
24. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Towards an Evaluation of Bidirectional Model-driven Spreadsheets. In: USER'12: User evaluation for Software Engineering Researchers. (2012) (to appear).
25. Cunha, J., ao P. Fernandes, J., Mendes, J., Pacheco, H., ao Saraiva, J.: Bidirectional transformation of model-driven spreadsheets. In Hu, Z., de Lara, J., eds.: Theory and Practice of Model Transformations. Volume 7307 of LNCS., Springer (2012) 105–120
26. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: MDSheet: A framework for model-driven spreadsheet engineering. In: ICSE'12: Proc. of the 34rd International Conference on Software Engineering, ACM (2012) 1412–1415
27. Cunha, J., Visser, J., Alves, T., Saraiva, J.: Type-safe evolution of spreadsheets. In: FASE'11/ETAPS'11: Proc. of the 14th International Conference on Fundamental Approaches to Software Engineering, Springer-Verlag (2011) 186–201
28. Cunha, J., Mendes, J., Fernandes, J.P., Saraiva, J.: Extension and implementation of classsheet models. In: VL/HCC '12: IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE Computer Society (2011) to appear.
29. Cunha, J., Saraiva, J., Visser, J.: Discovery-based edit assistance for spreadsheets. In: Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). VLHCC '09, Washington, DC, USA, IEEE Computer Society (2009) 233–237
30. Abraham, R., Erwig, M.: Inferring templates from spreadsheets. In: Proc. of the 28th Int. Conf. on Software Engineering, New York, NY, USA, ACM (2006) 182–191