# Automatically Inferring ClassSheet Models from Spreadsheets

Jácome Cunha*†
Universidade do Minho & ESTGF
jacome@di.uminho.pt

Martin Erwig
Oregon State University
erwig@eecs.oregonstate.edu

João Saraiva†
Universidade do Minho
jas@di.uminho.pt

## Abstract

*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. 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: (A) The benefit of creating a spreadsheet is lacking since the legacy spreadsheet already exists, and (B) existing data must be transferred into the new model-generated spreadsheet.*

*To address these problems and to support the model-driven spreadsheet engineering approach, we have developed a tool that can automatically infer ClassSheet models from spreadsheets. To this end, we have adapted a method to infer entity/relationship models from relational database to the spreadsheets/ClassSheets realm. We have implemented our techniques in the* HAEXCEL *framework and integrated it with the ViTSL/Gencel spreadsheet generator, which allows the automatic generation of refactored spreadsheets from the inferred ClassSheet model. The resulting spreadsheet guides further changes and provably safeguards the spreadsheet against a large class of formula errors. The developed tool 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.*

## 1. Introduction

Spreadsheets are one of the most popular programming systems, especially concerning business applications. Every year, hundreds of millions of spreadsheets are created by business users, and numerous studies show that this high rate of production is accompanied by an alarming high rate of errors [8, 12, 13]. Some studies report that up to 90% of real-world spreadsheets contain errors [14].

Spreadsheet systems offer users a high level of flexibility, making it easier for people to get started working with spreadsheets. The downside is that this freedom also offers ample opportunity to create erroneous spreadsheets. Errors during creation of a spreadsheet are made as well as when modified by other users. The problem gets exacerbated when the people who use or modify the spreadsheet do not fully understand its functionality. This situation arises because spreadsheet systems do not offer any higher-level abstractions like modern programming languages do. Moreover, data and computation are not separated in spreadsheets, and the immediate visual feedback mechanism makes traditional coding and program compilation/execution steps indistinguishable from each other. These factors make widespread reuse of spreadsheets difficult and prone to errors.

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 [2], ClassSheets [6], and the inclusion of visual objects to provide editing assistance in spreadsheets [5]. All these approaches aim at a form of model-driven software development: First, a spreadsheet business model is defined, from which then a customized spreadsheet application is generated that guarantees the consistency of the spreadsheet with the underlying model.

Despite of its huge 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 (business logic) and with model-driven software development. As some

*Supported by the Portuguese Research Foundation (FCT) PhD grant SFRH/BD/30231/2006.

†Supported by the SSaaPP project (SpreadSheets as a Programming Paradigm) under FCT contract PTDC/EIA-CCO/108613/2008.

978-0-7695-4206-5/10 $26.00 © 2010 IEEE
DOI 10.1109/VLHCC.2010.22

| <> | 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 | | 7.01 | 1 |
| 3 | 653 | 1111140009 | DOVE DISH LIQUID | 42 OZ | 9 | 2851281 | 100 | 384 | 7 | 1 | 2.19 | | 3.07 | 1 |
| 4 | 653 | 1111140009 | DOVE DISH LIQUID | 42 OZ | 9 | 2851281 | 100 | 385 | 15 | 1 | 2.19 | | 6.57 | 1 |
| 5 | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 6 | 654 | 1111165003 | SUNLIGHT AUTO GEL | 88 OZ | 6 | 2857061 | 100 | 390 | 6 | 1 | 3.75 | | 4.50 | 1 |
| 7 | 654 | 1111165003 | SUNLIGHT AUTO GEL | 88 OZ | 6 | 2857061 | 100 | 391 | 11 | 1 | 3.39 | S | 7.46 | 1 |
| 8 | 654 | 1111165003 | SUNLIGHT AUTO GEL | 88 OZ | 6 | 2857061 | 100 | 392 | 8 | 1 | 3.75 | | 6.00 | 1 |
| 9 | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

**Figure 1. A spreadsheet representing a sales system for dishwasher detergents.**

studies suggest, defining the business model of a spreadsheet can be a complex task for end users [1]. 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.

In this paper we propose reverse engineering techniques to derive ClassSheet models from existing spreadsheets. We use data mining techniques to reason about spreadsheet data and to infer functional dependencies among columns. Functional dependencies and a relational schema derived from them are the building blocks to infer the spreadsheet business model.

The rest of this paper is organized as follows. In Section 2 we present a motivating example. The algorithm for the automatic inference of ClassSheets from spreadsheets is explained in Section 3. An evaluation of the techniques presented can be found in Section 4. Related work is discussed in Section 5, and Section 6 concludes the paper.

## 2. An Example

Consider the example spreadsheet shown in Figure 1, taken from [13]. This spreadsheet represents a sales system for dishwasher detergents. The abbreviated labels have the following meanings: Label *com_code* stands for commercial code, *upc* for universal product code, *case* for number of cases, *move* for sales quantity, *qty* is a transaction quantity (which in this dataset is always set to 1), *ok* is a confirmation code, and *profit* is the profit and is calculated using the formula `=I2*K2*0.2` (for the first row).

The business logic that underlies this spreadsheet is not immediately clear and 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.

Objects that are contained in such a spreadsheet and the relationships between them are reflected by the presence of *functional dependencies* between spreadsheet columns. A functional dependency between a column $C$ and another column $C'$ means that the values in column $C$ determine the values in column $C'$, that is, there are no two rows in the spreadsheet that have the same value in column $C$ but differ in their values in column $C'$. This idea can be extended to multiple columns, that is, when any two rows that agree in the values of columns $C_1,\ldots,C_n$ also agree in their value in columns $C'_1,\ldots,C'_m$, then $C'_1,\ldots,C'_m$ are said to be *functionally dependent* on $C_1,\ldots,C_n$.

It is possible to construct a relational model from a set of observed functional dependencies. Such a model consists of a set of relation schemas (each given by a set of column names) and expresses the basic business model present in the spreadsheet. Each relation schema of such a model basically results from grouping functional dependencies together. For example, for the spreadsheet in Figure 1 we could infer the following relational model (underlined column names indicate those columns on which the other columns are functionally dependent).

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

The model has three relations: *StoreWeek* contains the data of stores and weeks of sale; *Detergent* contains all the information about detergents; *Sale* stores the information on sales, that is, for a particular detergent, in a certain store and week how much was sold, the profit, the price, if it was on sale or not, the quantity, and a confirmation code.

Although a relational model is very expressive, it is not quite suitable for spreadsheets since spreadsheets need to have a layout specification. In contrast, the *ClassSheet* modeling framework offers high-level, object-oriented formal models to specify spreadsheets and thus present a promising alternative [6]. ClassSheets allow users to express business object structures within a spreadsheet using concepts from the Unified Modeling Language (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 ClassSheet as the underlying modeling approach for spreadsheets and transform the inferred relational model into a ClassSheet model. In Figure 2 we present a new spreadsheet generated from the ClassSheet modeling our running example.

| | A | B | C | D | E | F | G | H | I | J | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | StoreWeek | | | | | | StoreWeek | | | | | |
| 2 | | Store | Week | | | | | Store | Week | | | | |
| 3 | | 100 | 383 | | | | | 100 | 384 | | | | |
| 4 | DishKey | | | | | | | | | | | | |
| 5 | Upc | Move | Price | Profit | Onsale | Qty | Ok | Move | Price | Profit | Onsale | Qty | Ok |
| 6 | 1111140009 | 16 | 2,19 | 7,01 | | 1 | 1 | 7 | 2,19 | 3,07 | | 1 | 1 |
| 7 | 1111165003 | 9 | 2,99 | 5,38 | | 1 | 1 | 21 | 2,99 | 12,56 | | 1 | 1 |
| 8 | | | | | | | | | | | | | |
| 9 | Detergent | | | | | | | | | | | | |
| 10 | Upc | Com_ code | Description | Size | Case | Nitem | | | | | | | |
| 11 | 1111140009 | 653 | DOVE DISH L | 42 OZ | 9 | 2851281 | | | | | | | |
| 12 | 1111165003 | 654 | SUNLIGHT AL | 88 OZ | 6 | 2857061 | | | | | | | |

**Figure 2. A spreadsheet generated from the ClassSheet modeling the running example.**

The underlying ClassSheet model can be divided in two main parts: the bottom one (rows 9 to 12) corresponds to the relation *Detergent*. It is a *class* composed of its name **Detergent**, by a row of labels corresponding to the attributes in the relational model, and a row with default values. In this instance, two rows (11 and 12) with values are shown.

The top part (rows 1 to 7) represents the sales relation. Since *Sale* is a relationship between two other relations, it is represented as a *cell class* (blue[1] tables) and the two related classes **Detergent** and **StoreWeek**. In fact, the table *Detergent* was already represented, and so only its key is used. The table *Detergent* was factored out in this way to avoid update anomalies, which occur when data is changed in one place and not changed in other places in the exactly same way [15]. This spreadsheet has two entries for sales.

ClassSheets carry information rich enough that allows the automatic generation of UML class diagrams from them. Figure 3 shows the UML class diagram that is derived from the ClassSheet from Figure 2.

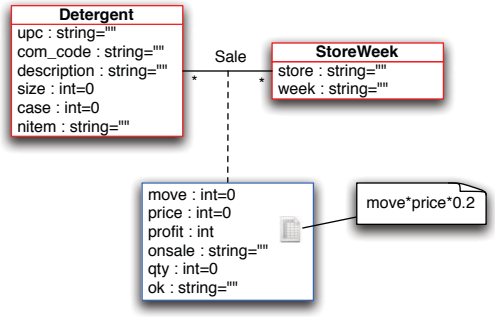


**Figure 3. A UML class diagram representing the business logic for the running example.**

In this section, we have described our approach to deriving ClassSheets for spreadsheets through an example. In fact, the ClassSheet and the UML class diagram were automatically produced by the tool that we have implemented based on our approach. In the coming sections we describe the steps and algorithms to infer ClassSheets in detail. The architecture of our approach is sketched in Figure 4.

[1]We assume that through the use of PDF this is visible to the reader.

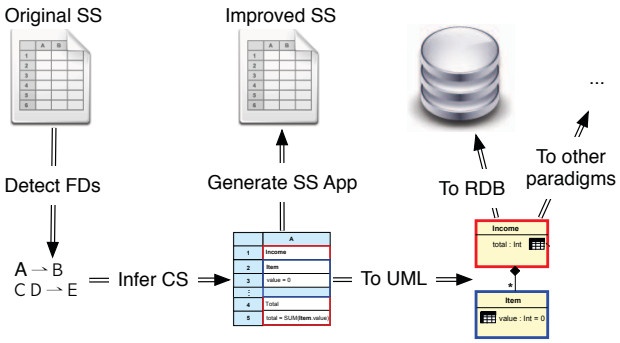


**Figure 4. An overview of the process to infer a ClassSheet and a UML class diagram.**

## 3. Inferring ClassSheets

In this section we explain in detail the steps to automatically extract a ClassSheet model from a spreadsheet. Essentially, our method involves the following steps:

(1) Detect all functional dependencies and identify model-relevant functional dependencies
(2) Determine relational schemas with candidate, foreign, and primary keys
(3) Generate and refactor a relational graph
(4) Translate the relational graph into a ClassSheet

In the following we will explain the steps 1, 3, and 4. Step 2 is a standard inference procedure borrowed from relational database theory [3].

### 3.1. Detecting Functional Dependencies

Knowledge about the functional dependencies in a spreadsheet provides the basis for identifying tables and their relationships in the data, which form the basis for ClassSheet models. The more accurate we can make this inference step, the better the inferred ClassSheet models will reflect the actual business models. In this subsection we will therefore first repeat some definitions regarding relational models and functional dependencies and then describe a set of heuristics that are key to inferring valid dependencies.

A (relational) schema is a finite set of attributes. In the context of spreadsheets an attribute corresponds to the label of a column, such as *upc* and *week*. A (relational) table over a schema is given by a finite set of tuples, where tuples correspond to (contiguous parts of) spreadsheet rows (or columns). Our example includes the tables *Detergent*

and *StoreWeek*. Each tuple is uniquely identified by a set of attributes called *primary key*. For example, in the *Detergent* table this set is $\{upc\}$. If a table has more than one set of attributes that can serve as primary key, called *candidate keys*, one of those has to be chosen as the primary key. A *foreign key* is a set of attributes within one table that matches the primary key of some other table.

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 $B$. For instance, in our running example the dependency *upc* $\rightharpoonup$ *description* exists, meaning that the values in column *upc* determine the values in column *description*.

Our goal is to use the data in a spreadsheet to identify functional dependencies. Depending on the data, it can happen that many "accidental" functional dependencies are detected, that is, functional dependencies that do not reflect the underlying model. For example, in our example we can identify the following dependency that is just happens to be fulfilled for this particular data set, but that does certainly *not* reflect a constraint that should hold in general: *size* $\rightharpoonup$ *com_code case qty ok*. In fact, the data contained in the spreadsheet example supports over 30 functional dependencies. Here are a few more.

*store* $\rightharpoonup$ *qty ok*
*com_code* $\rightharpoonup$ *qty ok*
*nitem* $\rightharpoonup$ *com_code size case qty ok*
*upc* $\rightharpoonup$ *com_code description size case nitem qty ok*
*move price* $\rightharpoonup$ *profit*

It should be noticed that the last functional dependency is induced by the formula `=I2*K2*0.2`, as described in [4].

The next step is therefore to filter out as much as possible those dependencies that are just accidental and keep the ones that are indicative of the underlying model. These can then be used to infer tables with primary and foreign keys.

The process of identifying the "valid" functional dependencies is, of course, ambiguous in general. Therefore, we employ a series of heuristics for evaluating dependencies.

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, in particular, through the spatial arrangement of cells and through labels [7].

In the following we describe the employed heuristics. Each of these can add support to a functional dependency.

**Label semantics** This heuristic is used to classify antecedents in functional dependencies. Most keys are labeled as "code" or "number" or are a combination of these labels with a label more related to the subject. For example, in a spreadsheet to store movies, a label "movie_id" could exist to uniquely identify a movie. We consider labels "id", "code", "number", "nr", "no" and combinations of them with other labels. A functional dependency with an antecedent of this kind receives high support.

**Label arrangement** If the functional dependency respects the original order of the attributes, this counts in favor of this dependency since very often key attributes appear to the left of non-key attributes.

**Antecedent size** Good primary keys often consist of a small number of attributes. Therefore, the smaller the number of antecedent attributes, the higher the support for the functional dependency.

**Ratio between antecedent and consequent sizes** In general, functional dependencies with smaller antecedents and larger consequents are stronger and thus more likely to be a reflection of the underlying data model. Therefore, a functional dependency receives the more support, the smaller the ratio of the number of consequent attributes is compared to the number of antecedent attributes.

**Single value columns** It sometimes happens that spreadsheets have columns that contain just one and the same value. In our example, the columns *ok* and *qty* are like this. Such columns tend to appear in almost every functional dependency's consequent, which causes them to be repeated in many relations. Since in almost all cases, such dependencies are simply a consequence of the limited data (or represent redundant data entries), they are most likely not part of the underlying data model and will thus be ignored.

After having gathered support through these heuristics, we aggregate the support for each functional dependency and sort them from most to least support. We then select functional dependencies from that list in the order of their support until all the attributes of the schema are covered.

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

*price* $\rightharpoonup$ *onsale*
*store week profit* $\rightharpoonup$ *move*
*upc* $\rightharpoonup$ *com_code description size case nitem*

One of the objectives of creating a ClassSheet model based on a relational model is to avoid update anomalies. To eliminate such problems, the dependencies must be normalized, for which we use an algorithm presented by Maier in [11].

Skipping over the details of that algorithm, we just note that in our example we obtain tables with only one candidate key each as shown in Table 1.

The final relational schema that we obtain is as follows.

*Price* ($\underline{price}$, *onsale*)
*StoreWeek* ($\underline{store, week, profit}$, *move*)
*Detergent* ($\underline{upc}$, *com_code*, *description*, *size*, *case*, *nitem*)
*Sale* ($\underline{upc, store, week, qty, price, profit}$, *ok*)

| Candidate Key Att. | | Foreign Key Att. | |
|---|---|---|---|
| Schema | Attribute | Schema | Attribute |
| Price | price | Sale | price |
| Detergent | upc | Sale | upc |
| StoreWeek | store | Sale | store |
| StoreWeek | week | Sale | week |
| StoreWeek | profit | Sale | profit |

**Table 1. Foreign keys for our example.**

Our tool currently assigns single letters as table names as default values. These can then be changed by the user (as we have done in this example) before the ClassSheet model is completed. Note that the relational model generated is not equal to the one presented in Section 2. From a data point of view the differences are not significant and the model serves the same purpose.

### 3.2. The Relational Intermediate Directed Graph

The next step in the reverse engineering process is to produce a *Relational Intermediate Directed (RID) Graph* [3]. This graph includes all the relationships between a given set of schemas. Nodes in the 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 with cardinality "*" at both ends is added to the graph.

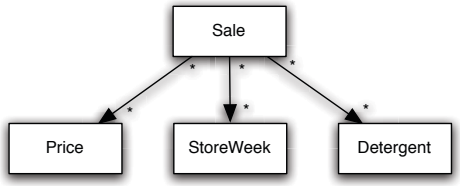


**Figure 5. RID graph for our running example.**

Figure 5 represents the RID graph for the Detergent sales system. This graph can generally be improved in several ways. For example, the information about foreign keys may lead to additional links in the RID graph. If two relations reference each other, their relationship is said to be *symmetric* [3]. One of the foreign keys can then be removed. In our example there are no symmetric references.

Another improvement to the RID graph is the detection of relationships, that is, whether a schema is a relationship connecting other schemas. In such cases, the schema is transformed into a relationship. The details of this algorithm are not so important and left out for brevity.

Since the only candidate key of the schema *Sale* is the combination of all the other schemas' primary keys, it is a relationship between all the other schemas and is therefore transformed into a relationship. The improved RID graph can be seen in Figure 6.

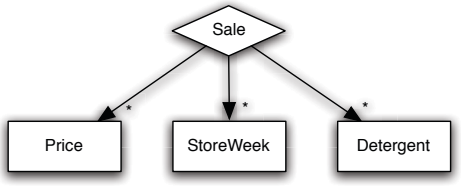


**Figure 6. Refactored RID graph.**

### 3.3. Generating ClassSheets

The RID graph generated in Section 3.2 can be directly translated into a ClassSheet diagram. By default, each node is translated into a class with the same name as the node and a vertically expanding block. In general, for a schema/node $A$ with attributes $A_1,\ldots,A_n$ and default values $da_1,\ldots,da_n$, a ClassSheet table as shown in Figure 7 is generated. (We omit here the column labels, whose names depend on the number of columns in the generated table.)

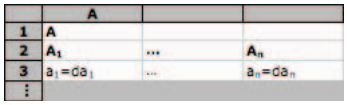


**Figure 7. Generated class for a node $A$.**

This ClassSheet represents a spreadsheet "table" with name **A**. For each attribute, a column is created labeled with the attribute's name. The default values depend on the attribute's domain. This table expands vertically.

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.

Relationships are treated differently and will be translated into cell classes. We distinguish between two cases: (1) Relationships between two schemas, and (2) relationships between more than two schemas. For the first case, let us consider the following set of schemas.

$$M(\underline{M_1,...,M_r},M_{r+1},...,M_s)$$
$$N(\underline{N_1,...,N_t},N_{t+1},...,N_u)$$
$$R(\underline{M_1,...,M_r,N_1,...,N_t,R_1,...,R_x},R_{x+1},...,R_y)$$

The corresponding RID graph is shown in Figure 8.

The ClassSheet that is produced by the translation is shown in Figure 9. For both nodes $M$ and $N$ a class is created as explained before (lower part of the ClassSheet). The


**Figure 8. RID graph for a binary relationship.**

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 value $dn_1$ in class **N**. This makes the spreadsheet easier to maintain while avoiding update anomalies. Class **Mkey** is created using the key attributes of the class **M** and the rest of the key attributes of the relation $R$. The cell class (with blue border) is created using the rest of the attributes of the relation $R$.

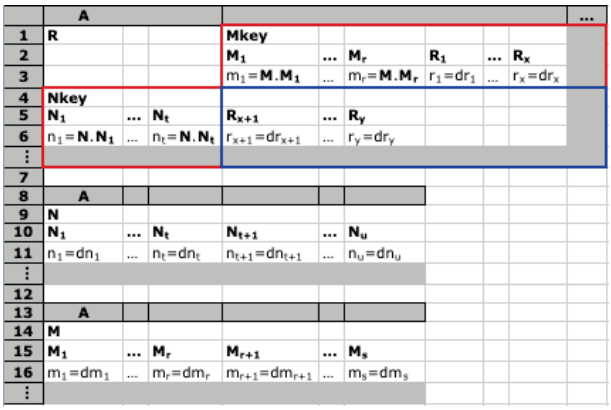


**Figure 9. ClassSheet for a relationship.**

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.

First, one of the relations **M** or **N** might have only key attributes. In this case, and since all the attributes of that class are already included in the cell class **R**, no separated class is created for it. The resultant ClassSheet would be similar to the one presented in Figure 9, but a separated class would be omitted for **M** or for **N** or for both.

The second case occurs when the key of the relation $R$ is only composed by the keys of $M$ and $N$, that is, $R$ is defined as follows: $R(\underline{M_1,...,M_r,N_1,...,N_t},R_1,...,R_x)$. (Part of) the resultant ClassSheet is shown in Figure 10.

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. (Although in this figure classes **M** and **N** do

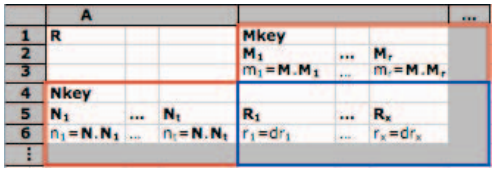


**Figure 10. ClassSheet for a relationship.**

not appear, they are generated in the same way as before.)

The third case occurs when the relationship is composed only of key attributes. Thus, the attributes that appear in the cell class are the non-key attributes of **N** and no class is created for **N**. This is the case of our example where **N** corresponds to *StoreWeek*, **M** to *Detergent* and **R** to *Sale*.

Finally, with more than three classes, 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 three relations, the generation proceeds as described above. The remaining relations are created in as explained in the beginning of this section.

In Figure 11 we present the ClassSheet model that is generated by our tool for the Detergent application.

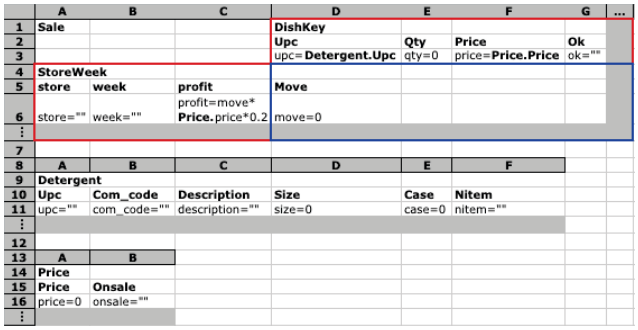


**Figure 11. The ClassSheet generated by our algorithm to the running example.**

We can observe that this model is different from the one that we have used for the spreadsheet instance in Figure 2, which illustrates that ClassSheet models are not unique, and which also raises the question about the quality of models inferred by our tool. We address this question next.

## 4. Evaluation

In order to evaluate the applicability of our approach, we have performed an experiment on the spreadsheets that are made available (through a CD) with [13]. This set consists of 27 spreadsheets, each containing between 1 and 16 work-

sheets, 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?

### 4.1. 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 2 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 2 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 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 model 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 ClassSheet models fall under this classification.

The classification *acceptable* was given to ClassSheet models 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 ClassSheet models as *acceptable*.

Finally, a *good* ClassSheet model 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 ClassSheet models.

| Spreadsheet | sheets/tables | | fail | bad | acc | 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 2. Results of our evaluation.**

### 4.2. Discussion

The test results are quite encouraging: With our techniques we are able to produce ClassSheet models 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 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 much more effective if it was helped by a human. For example, if additional dependencies or headers

[2]One spreadsheet, c6/Adbudget6.xls, was unreadable.

were explicitly provided by the user, the generated models could be improved. Therefore, an extension of our tool allowing users to interact and provide input about the models seems to be the most important direction for future work.

## 5. Related Work

The process of inferring models from existing spreadsheets has been proposed before. Extracting templates from existing spreadsheets was described in [1]. That technique 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.

Hermans et al. describe a technique to automatically infer class diagrams for existing spreadsheets matching them with a set of pre-defined spreadsheet patterns [9]. This technique works very well for spreadsheets following the predefined patterns. This approach does not consider other type of patterns (like the data dependencies we compute), and the derived model is not as rich as the ClassSheet one.

Isakowitz et al. proposed a methodology to infer a logical layer from existing spreadsheets [10]. This process in not completely automatic and sometimes requires human intervention. The reverse process is also presented: to build a spreadsheet application reusing the logical models inferred. To separate the model from the data is a concern of both Isakowitz and our work.

ClassSheets and entity-relationship (ER) diagrams are closely related. There have been proposed several algorithms to infer ER diagrams from databases. We have reused some algorithms proposed by Alhajj in [3] to help us construct an intermediate model of a spreadsheet.

A transformation of spreadsheets to the relational setting was proposed by Cunha et al. in [4]. Based on functional dependencies a schema is generated and a new spreadsheet based on that model is produced. The model is based only on database techniques and is not well adapted to spreadsheets, in particular, the identification of dependencies ignores the peculiarities of spreadsheets.

## 6. Conclusions

We have developed a technique and a tool that can automatically infer ClassSheets from spreadsheets. These models can be of great help for the maintenance of spreadsheets since knowledge about the underlying model can, for example, prevent erroneous operations. We have adapted and extended a method to infer entity/relationship models from relational databases to work with spreadsheets. The exploitation of layout information that is specific to spreadsheets was instrumental in the successful working of the tool. An evaluation of our tool on real-world spreadsheets showed encouraging results and demonstrated that the approach is viable and should be pursued further in future work.

## References


[1] R. Abraham and M. Erwig. Inferring templates from spreadsheets. In *ICSE'06: Proc. of the 28th Int. Conf. on Sof. Eng.*, pages 182–191, New York, NY, USA, 2006. ACM.

[2] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual specifications of correct spreadsheets. In *VLHCC '05: Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 189–196, Washington, DC, USA, 2005. IEEE Computer Society.

[3] R. Alhajj. Extracting the extended entity-relationship model from a legacy relational database. *Information Systems*, 28(6):597 – 618, 2003.

[4] J. Cunha, J. Saraiva, and J. Visser. From spreadsheets to relational databases and back. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 179–188, New York, NY, USA, 2008. ACM.

[5] J. Cunha, J. Saraiva, and J. Visser. Discovery-based edit assistance for spreadsheets. In *VLHCC '09: Proc. of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 233–237. IEEE Computer Society, 2009.

[6] G. Engels and M. Erwig. ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In D. Redmiles, T. Ellman, and A. Zisman, editors, *20th IEEE/ACM Int. Conf. on Automated Sof. Eng., Long Beach, USA*, pages 124–133. ACM, 2005.

[7] M. Erwig. Software Engineering for Spreadsheets. *IEEE Software*, 29(5):25–30, 2009.

[8] EuSpRIG. European spreadsheet risks interest group. `http://www.eusprig.org/`.

[9] F. Hermans, M. Pinzger, and A. van Deursen. Automatically extracing class diagrams from spreadsheets. In *Proc. 24th European Conf. on Object-Oriented Programming (ECOOP)*, 2010.

[10] T. Isakowitz, S. Schocken, and H. C. Lucas, Jr. Toward a logical/physical theory of spreadsheet modeling. *ACM Trans. Inf. Syst.*, 13(1):1–37, 1995.

[11] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[12] R. Panko. Spreadsheet errors: What we know. What we think we can do. *Proc. of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.

[13] S. G. Powell and K. R. Baker. *The Art of Modeling with Spreadsheets*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[14] K. Rajalingham, D. Chadwick, and B. Knight. Classification of spreadsheet errors. *Proc. of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (EuSpRIG)*, 2001.

[15] J. D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, 1997.