Design and Implementation of Queries for Model-Driven Spreadsheets*

Jácome Cunha[1,2], João Paulo Fernandes[1,3], Jorge Mendes[1], Rui Pereira[1], and João Saraiva[1]

[1] HASLab/INESC TEC & Universidade do Minho, Portugal
[2] CIICESI, ESTGF, Instituto Politécnico do Porto, Portugal
[3] RELEASE, Universidade da Beira Interior, Portugal
{jacome, jpaulo, jorgemendes, ruipereira, jas}di.uminho.pt

Abstract. This paper presents a domain-specific querying language for model-driven spreadsheets. We briefly show the design of the language and present in detail its implementation, from the denormalization of data and translation of our user-friendly query language to a more efficient query, to the execution of the query using Google. To validate our work, we executed an empirical study, comparing *QuerySheet* with an alternative spreadsheet querying tool, which produced positive results.

Keywords: spreadsheets, model-driven engineering, querying

1 Introduction

Nowadays, spreadsheets can be considered one of the most popular programming system around, particularly in the field of business applications, and one of the largest domain specific programming languages. With their availability on any computing device (PC, smart-phone, etc.) and in the cloud, visual simplicity, low learning curve for new users, and flexibility when it comes to what can be written in a spreadsheet, the amount of users per year increases drastically. Although spreadsheets begin as a simple, single-user software artifact, they may evolve into a large and complex data-centric software [1]. In these cases, manipulating a large amount of data in a traditional matrix structure becomes an arduous task. This issue arises in spreadsheets, unlike the traditional database systems, due to one considerable flaw: the absence of a data query language.

The problem of querying data is not new, having decades worth of attention within the database community. Yet, only recently has it been seriously considered in the context of spreadsheets. And even then, these attempts to replicate a

*This work is part funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within projects `FCOMP-01-0124-FEDER-010048`, and `FCOMP-01-0124-FEDER-022701`. The first author was funded by FCT grant `SFRH/BPD/73358/2010`.

traditional database querying system have several drawbacks of their own. Most impose restrictions on how the data must be stored, organized, and represented, and some even have a hard-to-read query language.

To solve these problems, we propose a query language based on the Structured Query Language (SQL) where users can easily construct queries right in their spreadsheet environment, without the need of complicated configurations, or extra programs other than a simple add-on. Both SQL and spreadsheets can be seen as domain-specific functional programming languages [2]. Our approach builds upon a model-driven spreadsheet development environment, where the queries would be expressed referencing entities in ClassSheet models, instead of the actual data, allowing the user to not have to worry about the arrangement of the spreadsheet's data, but only what information is present.

This allows spreadsheet evolution to occur in the data or the arrangement of entities within a spreadsheet model, without invalidating previously constructed queries, as long as the entities continue to exist. The query results are then shown as an inferred spreadsheet model, and a new worksheet in conformance with the model. This system was named *QuerySheet* [3], and will be shown further on.

Our goal is to make spreadsheet querying more humanized, understandable, robust, and productive. In order to validate our achievements, we executed an empirical study with real end users. Their experiences, productivity, and feedback in using the *QuerySheet* system were recorded and are now presented.

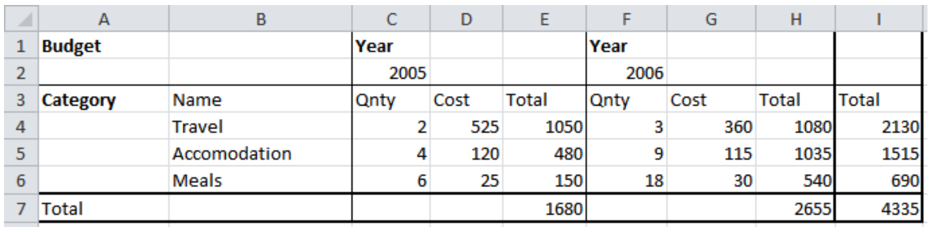
The results observed from this study were positive, as we will discuss further on. Also, we plan to take on the user's feedback to further improve our framework.

This paper is organized as follows: Section 2 presents existing techniques to query spreadsheets, detailing two specific approaches. In Section 3, a simple introduction to model-driven spreadsheets is given. Section 3 explains the spreadsheet querying system we propose, and shows an example of that envisioned system. In Section 4 we present queries for model-driven spreadsheets. We then present in Section 5 the design and implementation of our model-driven spreadsheet system, along with a small demonstration of the actual tool in Section 6. Section 7 details our empirical study and presents the results. And finally, Section 8 presents our concluding thoughts and future work.

2 Spreadsheets and Queries

Before we present techniques to query spreadsheets, let us introduce a spreadsheet to be used as a running example throughout this document. Figure 1 presents a spreadsheet to store information about the budget of a company. This spreadsheet contains information about the Category of budget use (such as Travel or Accommodation) and the Year. The relationship between these two entities gives us information on the Quantity, the Cost, and the Total Costs.

As previously stated, there have been attempts to query spreadsheets using some form of SQL. Two widely known names followed this path to create a spreadsheet querying system: Microsoft and Google, with their MS-Query Tool



	A	B	C	D	E	F	G	H	I
1	**Budget**		**Year**			**Year**			
2			2005			2006			
3	**Category**	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total
4		Travel	2	525	1050	3	360	1080	2130
5		Accomodation	4	120	480	9	115	1035	1515
6		Meals	6	25	150	18	30	540	690
7	Total				1680			2655	4335

Fig. 1. Spreadsheet data for a Budget example

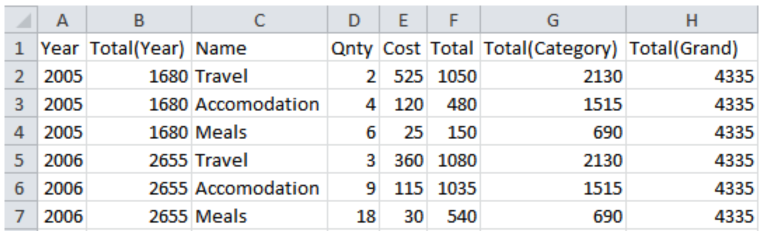
and Google QUERY Function respectively. The following subsections will give a brief description of each of these approaches.

2.1 MS-Query Tool

Microsoft's Query tool, or MS-Query, is the database query interface used by Microsoft Word and Excel, a utility which imports databases, text files, OLAP cubes, and other spreadsheet representations (such as csv). While these are the main uses, it can be used to query data from a spreadsheet, placing the data into an intermediate database-like table to be able to apply the query and represent the findings, but in turn brings some restrictions.

To be able to query the spreadsheet data, the data itself must be in a single tabular format, with the headers present in the first row. In other words, they require the data to be denormalized [4] if the user wishes to completely represent his spreadsheet information. In most cases, users tend to use their spreadsheet for more than one entity in a single worksheet, not joining all the information into one single unified table (as we can see in our running example in Figure 1). This requirement prohibits the freedom to represent the spreadsheet data how a user wishes.

Figure 2 shows the necessary denormalized representation of the data in our running example, having the headers of each attribute explicitly represented in a single row, just so we may be able to query the data using the MS-Query tool.



	A	B	C	D	E	F	G	H
1	Year	Total(Year)	Name	Qnty	Cost	Total	Total(Category)	Total(Grand)
2	2005	1680	Travel	2	525	1050	2130	4335
3	2005	1680	Accomodation	4	120	480	1515	4335
4	2005	1680	Meals	6	25	150	690	4335
5	2006	2655	Travel	3	360	1080	2130	4335
6	2006	2655	Accomodation	9	115	1035	1515	4335
7	2006	2655	Meals	18	30	540	690	4335

Fig. 2. Spreadsheet data for a Budget example (denormalized)

As one may notice, the representation of the data in this way is much harder for someone to read, manage, and analyze, and if looking at a real-life spreadsheet, which might have the number of columns reaching the hundreds, it can become even more difficult. Along with the previously mentioned problem, a user with this representation, may not expand his information horizontally, but only vertically, to conform to the table format needed to query, allowing even less freedom to represent the data.

2.2 Google QUERY Function

Google provides a QUERY function (GQF) which allows users, using a SQL-like syntax, to perform a query over an array of values. An example would be their Google Docs spreadsheets, where the function is built-in.

In this setting, a query is a two part function, consisting of a *range* as its first argument, to state the range of the data cells to be queried, for example A1:B6. The second part consists of the *query string* itself, using a subset of the SQL language, with column letters. The function's input also assumes the first row as headers, and each column of the input can only hold values of certain types. An example function is shown in Listing 1.1. This function can actually be written on the spreadsheet itself, allowing on-the-spot results.

Listing 1.1. Google QUERY function example

```
=query(A1:F53;"SELECT A, B, F WHERE D > 5")
```

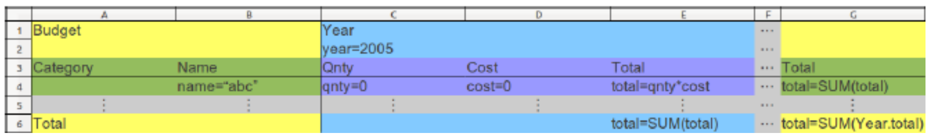
While being a powerful query function, it still has its flaws. The function shares the same problems as MS-Query in regards to the data representation. Much like MS-Query, to run the function, the data needs to be represented with a single header row, without relationships between the entities, in other words, also denormalized (as already shown in Figure 2).

Along with the difficulty of managing the data in such a way, the function has another flaw. Instead of writing the query using column names/labels, one must use the column letters (as shown in Figure 1.1) to write the query. Even with the small sized example we have been using, column letters and not names can get confusing, counter-intuitive, and almost impossible to understand what the query is supposed to do, without having the data sheet alongside. Moreover, Google queries do not truly support evolution, since they do not adapt/evolve when the spreadsheet data evolves. That is to say, by adding a new column to the spreadsheet, we may turn a query invalid or incorrect because the data changed positioning in the spreadsheet.

3 Model-Driven Spreadsheet Engineering[4]

To overcome the issues identified in Section 2, and to design a language and system which match the previously defined criteria, we turned to model-driven engineering methodologies [5, 6]. Model-driven engineering is a development methodology in software development that uses and exploits domain models, or abstract representations of a piece of software, a solution to the handling of complex and evolving software systems. This has been applied to spreadsheets, making model-driven spreadsheets possible [7, 8], and even a model-driven spreadsheet environment [9, 10].

One of these spreadsheet models is ClassSheets [11, 12], a high-level and object-oriented formalism, using the notion of classes and attributes, to express business logic spreadsheet data. Using ClassSheets, we can define the business logic of a spreadsheet in a concise and abstract manner. This results in users being able to understand, evolve, and maintain complex spreadsheets by just analyzing the (ClassSheet) models, avoiding the need to look at large and complex data.



	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6	Total				total=SUM(total)	⋯	total=SUM(Year.total)

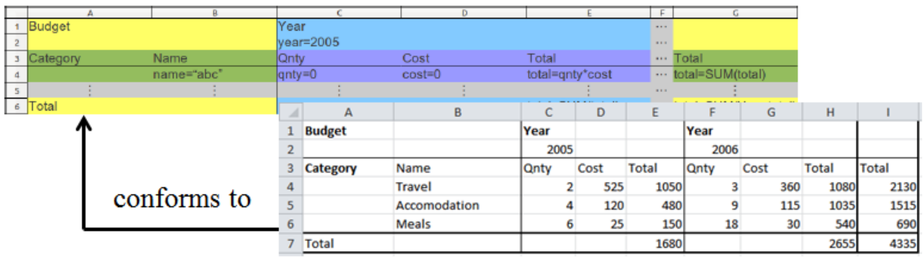
Fig. 3. ClassSheet model for a Budget example

To showcase ClassSheets, we present in Figure 3 a ClassSheet model for the Budget example shown in Figure 1. In this ClassSheet model, a **Budget** has a **Category** and **Year** class, expanding vertically and horizontally, respectively. The joining of these gives us a **Quantity**, a **Cost**, and the **Total** of a **Category** in a given **Year**, each with its own default value. The **Total** in column G gives us the total of each **Category** and the **Total** in column A gives us the total of each **Year**.

This ClassSheet model specifies the business logic of the budget spreadsheet data from our running example. In model-driven engineering, we would say that the spreadsheet data (Figure 1) *conforms to* the model (Figure 3), as shown in Figure 4.

Using models, we can also have a safe way to practice software evolution [13], a term defining the process of changing an existing software system or program, due to needs, rules, and other factors, is updated, or in other words evolves, to continue to be useful in its environment. Evolution, and other techniques

[4]ClassSheets are presented in great detail in a tutorial of DSL'13. If both papers are accepted, we may shorten this section and refer to the other paper.



	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6	Total						

	A	B	C	D	E	F	G	H	I
1	**Budget**		**Year**			**Year**			
2			2005			2006			
3	**Category**	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total
4		Travel	2	525	1050	3	360	1080	2130
5		Accomodation	4	120	480	9	115	1035	1515
6		Meals	6	25	150	18	30	540	690
7	Total				1680			2655	4335

Fig. 4. Spreadsheet model and example in conformity

(such as spreadsheet model embedding and bidirectionality) are present in the model-driven spreadsheet framework MDSheet [10, 14].

4 Model-Driven Spreadsheet Querying

Querying spreadsheets should be simple and intuitive as it is in the database realm. Using a simple SQL-like query language, users should be able to easily construct queries right in their spreadsheet environment, without the need of complicated configurations or extra programs. This language should be humanized, avoiding the use of computer-like terms as column letters, and use some form of labels or descriptive tags to point to attributes and entities. In fact, this is in line with the results presented in [15] where authors showed that spreadsheet users create a mental model of the spreadsheet that helps them understand and work with the spreadsheet. These mental models are created using names from the real world as it is the case with our ClassSheet models.

To do this, a good approach would be to build upon a model-driven spreadsheet development environment, where we can take advantage of ClassSheets, allowing the queries to be expressed referencing the entities in ClassSheet models instead of the data's positioning. This would allow the user to not have to worry about the arrangement of the spreadsheet's data, but only what information is present. This is almost identical to how a database administrator or analyst would look at the relational model of a database to construct queries, and not the data itself.

4.1 Querying Model-Driven Spreadsheets: An Example

To show how we envision the model-driven spreadsheet querying system, we will show an example.

Using our previous Budget model from Figure 3, and the Budget data from Figure 1, we will try to answer two simple questions:

Query1:	What was our budget use in 2005?
Query2:	What was our total quantity per year?

By simply looking at our ClassSheet model, if we do not remember or know the structure of our spreadsheet data, we would be able to write the following simple SQL-like queries:

Listing 1.2. Model-driven query for Query1

```
SELECT Name, Qnty, Cost, Total
WHERE Year = 2005
```

Listing 1.3. Model-driven query for Query2

```
SELECT Year, Sum(Qnty)
GROUP BY Year
LABEL Sum(Qnty) "Total Qnty"
```

These queries will produce the following results:

Table 1. Results for Query1

Name	Qnty	Cost	Total
Travel	2	525	1050
Accommodation	4	120	480
Meals	6	25	150

Table 2. Results for Query2.

Year	Total Qnty
2005	12
2006	30

The two equivalent Google QUERY functions would be Listing 1.4 and Listing 1.5 respectively:

Listing 1.4. Google QUERY function for Query1

```
=query(A1:H7;"SELECT C, D, E, F WHERE A = 2005")
```

Listing 1.5. Google QUERY function for Query1

```
=query(A1:H7;"SELECT A, sum(D)
GROUP BY sum(D) LABEL sum(D) 'Total Qnty'")
```

Using this model-driven approach, we eliminate the work of using column letters when writing the query, and the need of restricting the user's data to a specific format. This way, the user can maintain the original spreadsheet, without having to conform to data representation restrictions, and analyze it with references to the entities and attributes presents.

So as one can see, our approach hopes to make querying spreadsheets more:

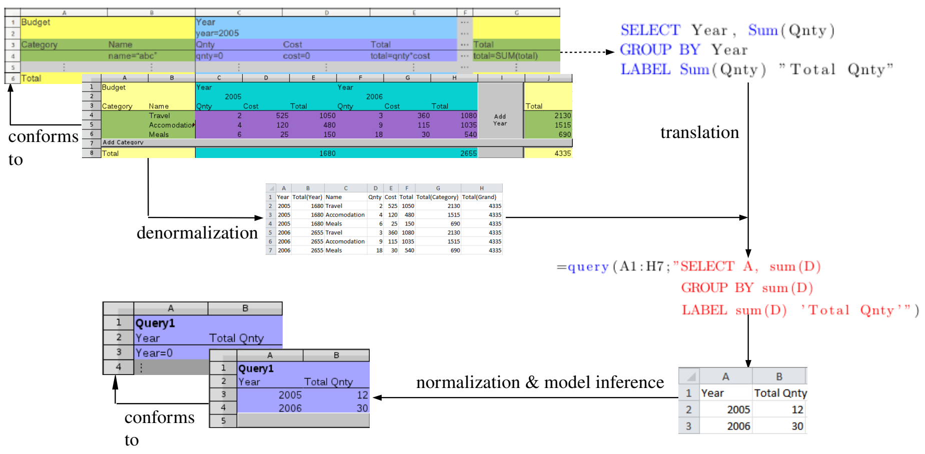
– *Humanized* - Now we can represent attributes and data areas (models) using human designated names, instead of column letters.

– *Understandable* - Now we can actually understand and easily read the queries, knowing exactly what they do.
– *Robust* - Unless attributes in the query are removed or renamed, the queries can still correctly function even with spreadsheet data/model evolutions.
– *Productive* - No need to manually think through what spreadsheet area our data is inn, or what column letter is a given attribute.

These four topics are what we strived to achieve. To validate these topics, we executed an empirical study, which is presented in Section 7 with more details.

5 Design and Implementation

In this section we explain how the model-driven query language system we envisioned has been materialized. Figure 5 presents the overall architecture of our system which we have implemented on top of MDSheet [9].



```
SELECT Year, Sum(Qnty)
GROUP BY Year
LABEL Sum(Qnty) "Total Qnty"
```

	A	B	C	D	E	F	G	H
1	Year	Total(Year)	Name	Qnty	Cost	Total	Total(Category)	Total(Grand)
2	2005	1680	Travel	2	525	1050	2130	4335
3	2005	1680	Accomodation	4	120	480	1515	4335
4	2005	1680	Meals	6	25	150	690	4335
5	2006	2655	Travel	3	360	1080	2130	4335
6	2006	2655	Accomodation	9	115	1035	1515	4335
7	2006	2655	Meals	18	30	540	690	4335

```
=query(A1:H7;"SELECT A, sum(D)
              GROUP BY sum(D)
              LABEL sum(D) 'Total Qnty'")
```

	A	B
1	Year	Total Qnty
2	2005	12
3	2006	30

Fig. 5. The model-driven query system

In MDSheet all mechanisms to handle models and instances are already created. This is our starting point: in the left part of the figure we show a spreadsheet instance and its corresponding model. The second required part is the query over the model/instance. This will be explained in detail in the next Subsection 5.1. The spreadsheet instance is then denormalized, as we will explain in Subsection 5.2, and the query over the model is translated into a Google query, as explained in Subsection 5.3. The Google query and the denormalized data are sent to Google and the result received is shown in the bottom-right part of the figure, described in Subsection 5.4. Finally, a new model is inferred so the result can be used as input to a new query, as explained in Subsection 5.5. This last step is necessary since we want the queries to be composable, and new models to be generated from queries.

Before presenting the algorithms that are used by our query mechanism, let us introduce our Haskell representation of models and instances.

```
data Class = Class Name Expansion [Attribute] HName VName
data Expansion = Horizontal | Vertical | Both | None
data Attribute = Attribute Name [Value]
data Value = Value Val InstanceH InstanceV

data Layout = Layout [(Name, ClassName)]
```

These are the four main data types for querying in our framework. The *Class* data type holds the *Class*'s *Name*, *Expansion* direction (either *Horizontal*, *Vertical*, *Both*, or *None*) a list of *Attribute*(s), and its horizontal/vertical class name if it has one (*HName*/*VName*). Each attribute also has a *Name*, and a list of *Values*, in which each Value has the value in a cell (*Val*), and its horizontal and vertical instance (*InstanceH* and *InstanceV*). These instances are used to know which relational classes to combine with. The *Layout* data type has the header information of the denormalized data, including the attribute's *Name* and class name (*ClassName*).

We can show the top function of our system. It receives the user's query, along with the worksheet being used. It then passes through all the processes previously mentioned (denormalization, translation, execution, and inference) and returns the new model and instance.

```
querysheet :: Query → Worksheet → (Model, Instance)
querysheet query worksheet =
  let (model, inst)              = getModelInstance worksheet
      (denormData, layout)       = denormalize model inst
      googleQueryFun             = translate query layout
      queryResults               = runGQF googleQueryFun denormData
      (newModel, newInstance)    = inferClassSheet queryResults
  in (newModel, newInstance)
```

In the next Subsections we will explain in more detail each of the steps of our algorithm.

5.1 Model-Driven Query Language

The Model-Driven Query Language (MDQL) is very similar to the standard SQL language, while also allowing some of the GQF's clauses such as LIMIT and LABEL. To create the MDQL, we used advanced engineering techniques, namely generalized top-down parsers and strategic programming to traverse trees.

The syntax of our query language is defined in the grammar shown in Listing 1.6. As we can see, instead of selecting column letters in the SELECT clause, the user can select the ClassSheet attributes he/she wishes to query, while also allowing him/her to further specify, as to avoid any naming conflicts which may occur, alternative ways of naming the attribute such as:

– stating its name - (**Cost**)
– stating the attribute along with its classes' name (**Year.Total**)
– stating both classes (**(Year, Category).Total** or **(Category, Year).Total**)
– stating all the attributes in a given class (**Category.***)

Listing 1.6. Part of the model-driven query language syntax

```
SELECT [DISTINCT] (* | attr1, ...,agg(attrX), ...)
 [FROM ClassSheet1, [JOIN ClassSheet2], ...]
 [WHERE conditions]
 [GROUP BY attr1, ...]
 [ORDER BY attr1 [ASC|DESC], ...]
 [LIMIT numRow]
 [LABEL attr1 'new_attr1', ...]
 [WITH HISTOGRAM]
attr ::= attribute
      | Class.*
      | (Class1, Class2).*
      | Class.attribute
      | (Class1, Class2).attribute
agg ::= Sum(attr)
      | Count(attr)
      | Avg(attr)
      | Min(attr)
      | Max(attr)
conditions ::= attr logic attr
             | attr logic 'string'
             | attr logic number
logic :: = < | > | <= | >= | == | !=
```

The MDQL also has a FROM clause, very reminiscent from the same clause in SQL, which allows the user to choose which ClassSheet model(s) to use for the query, in cases where more than one ClassSheet is present in a spreadsheet. Also note that as in SQL we allow JOIN operations between two ClassSheets (nonexistent in the GQF). We also have the LIMIT clause to limit the amount of results returned by a given number, and LABEL to rename attributes to a given name, both originating from GQF clauses. The WHERE, GROUP BY, and ORDER BY clauses work the same as in SQL, applying filters such as where an attribute is equal to a given name (e.g. Category.Name = 'Travel'), grouping values to apply an aggregation function, and ordering by a given attribute either ascendant (ASC) or descendant (DESC), all three respectively. Finally, the DISTINCT clause was also implemented (also nonexistent in the GQF) to remove duplicated results which may occur, and WITH HISTOGRAM is used to state if the user wishes the results produce a histogram chart to visually show the results.

Since this language is very similar to SQL, it allows users who already know basic SQL to simply jump into query writing in this system, avoiding the need to learn a new language, allowing us to adapt the most used query language instead of creating one, while also allowing queries to be more elegant, concise, robust and understandable for spreadsheets, along with being easy to learn since the SQL-language is often described as "English-like" because many of its statements read like English [16].

5.2 Denormalization of Spreadsheet Data

As mentioned before, to be able to use the Google Query Function, the data must be in a single matrix format, with the headers present in the first row. In consequence of this restriction, for a user to be able to write all the queries possible with the data, every bit of data from the spreadsheet has to be written in this single matrix structure. To do so, the data has to be in a redundant state, combining the data from multiple tables together, reminiscent of a JOIN between tables in databases, thus duplicating the data . In other words, we have to denormalize our spreadsheet data [17].

To correctly do this, we must first obtain all the necessary and critical information from the ClassSheet models, and their attributes/data. To begin, we obtain this information from the MDSheet framework (atleast in this context), such as which ClassSheet classes exist, their names, their expansion direction (horizontally, vertically, both, or none at all) and most importantly the attributes in each class.

After obtaining the ClassSheet models and data, we begin the denoraml-ization process, where we denormalize the models used in the query, and join the relational models with their corresponding horizontal and vertical classes. A fragment of that denormalization process can be seen next.

```
denormalize :: Model → Instance → (Data, Layout)
denormalize model inst =
  let allClasses      = merge model inst
      relationClasses = findRelations allClasses
      res             = relationDenorm relationClasses allClasses
      ssdata          = getData res
      layout          = getLayout res
  in (ssdata, layout)
```

As we can see, the first step is to merge the model and instance information together into an intermediate representation we use. Using that intermediate representation, we find the relational classes, for example (Category, Year), and then denormalize the data in the relation, and obtain the spreadsheet data and layout. The true process of denormalizing the data is presented next.

```
relationDenorm :: [Class] → [Class] → Table → Table
relationDenorm [] ac tab = tab
relationDenorm ((Class n exp attrs hName vName) : cs) ac tab =
  let hClass = getClass hName ac
      vClass       = getClass vName ac
      classResJoin = rJoin (Class n, exp, attrs, hName, vName) hClass vClass
      tabRes       = addTable classResJoin tab
      table        = relationDenorm cs ac tabRes
   in table

rJoin :: Class → Class → Class → Class
rJoin (Class n exp (attr : as) hName vName) hClass vClass =
  let hAttrs = getHInstances attribute (getAttributes hClass)
      vAttrs = getVInstances attribute (getAttributes vClass)
      clas = (Class n exp ((attr : as) ++ hAttrs ++ vAttrs) hName vName)
  in clas
```

We obtain, through the class names, the appropriate classes, which we then use to correctly match and join the information from the relational classes. This process happens in the *rJoin* function, where we use the *HInstances* and *VInstances* to properly match the relational class, with its two "parent" classes.

A more detailed explanation of the denormalization process, along with examples, and description of certain problems automatically solved, can be found in [14].

5.3 Translation to Google Query

The main reason we chose not to develop a new querying engine, but re-utilize the QUERY function's querying engine, is because we do not want to try to compete with Google in terms of performance and speed where Google has shown dominance in when developing querying engines.

To properly run the GQF, our model-driven queries must adhere to the *Visualization API Query Language* [18], specified by Google. So, for our model-driven queries to function correctly, a translator was made to transform the model-driven queries to their equivalents for the GQF. To do so, we took advantage of a strategy language to control transformations and pattern matching, to translate and inspect the query respectively.

The translator automatically calculates the *range* from the ClassSheet models selected, in the FROM clause for example, by using a lookup function to find what is the new range of data after the denormalization process. It also substitutes the attribute names to their corresponding column letters in the denormalized data, without the user having to do so. After parsing the user's query, and verifying that each attribute chosen by the user exists, and has no conflicts, such as any ambiguous attribute names due to the attribute name repeating in more than one ClassSheet (which may be solved by adding the class name beforehand as shown in 5.1), we apply another lookup function on each attribute, and calculate the column letter corresponding to each attribute. A fragment of one of the lookup functions (for translating an attribute with its class name) can be seen in the following:

```
lookUp :: (Name, ClassName) → Layout → String
lookUp p (Layout l) =
  let allIndices = elemIndices p l
  in if (length allIndices) ≡ 1
       then intToColumn (head allIndices)
       else "ERROR"
```

Using the *lookUp* function, we find the matching header, and if there is one and only one occurrence, we translate the index number to its appropriate column letter (for example 0 = A, AA = 27). If more than one occurrence occurs, or no occurrences, we send an error.

Now having both the denormalized data and translated model-driven query ready, we can send the spreadsheet data to Google Spreadsheets, run the GQF and afterwards retrieve the results for the user to view in its spreadsheet.

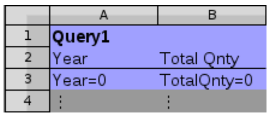
5.4 Google Spreadsheets

To be able to send the spreadsheet data to Google Spreadsheets and run the GQF, we turned to the *Google Spreadsheets API version 3.0* [19], an API which enables developers to be able to create applications that can read, write and modify the data in Google Spreadsheets. It allows us to manage the worksheets in a Google spreadsheet, manage cells in a worksheet by position, and also allows us to create spreadsheets, worksheets, insert and delete data, and retrieve a single worksheet or a spreadsheet, along with authorizing requests and authentication.

So before we acquire the query results, we begin by creating a temporary worksheet which will be filled with the denormalized data, followed by creating a second temporary worksheet where the query function string is sent to. When the query function is inserted into a cell, it calculates the results, and now that second worksheet contains the query results. Finally, the results are retrieved, the temporary worksheets removed and an inference technique is ran before presenting it to the user.

5.5 ClassSheet Inference

In order to make the queries composable, that is, to allow the output of a model-driven query as the input of another model-driven query we must provide the results from the GQF with a model. Without having a model, it is impossible to make a query on a result of another query. Previous work in this field introduced a technique to automatically infer a ClassSheet model from spreadsheet data [20]. Thus, applying this technique on the results obtained from the GQF, we can now infer the correct ClassSheet model and have it alongside the queried results. For example, applying the inference technique to the results from Query2 presented in Table 2, we would obtain the ClassSheet model shown in Figure 6, and now present the user the results alongside its model.

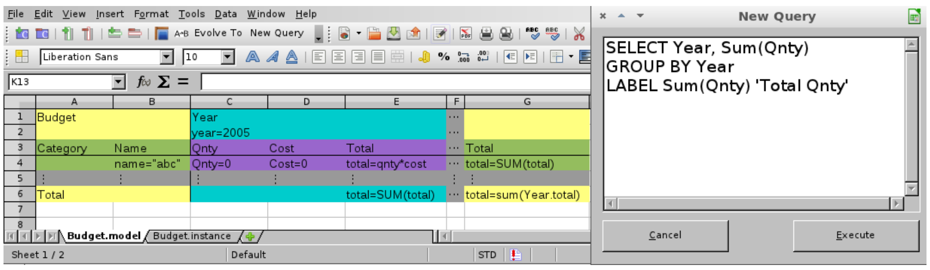


	A	B
1	**Query1**	
2	Year	Total Qnty
3	Year=0	TotalQnty=0
4	⋮	⋮

Fig. 6. Model automatically inferred from the spreadsheet data shown in Table 2

6 QuerySheet

The model-driven query language and the techniques proposed in the previous sections are the building blocks used to construct a tool, integrated in MDSheet and OpenOffice/LibreOffice, named *QuerySheet* [3].



	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	Total
4		name="abc"	Qnty=0	Cost=0	total=qnty*cost	⋯	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋮	⋮
6	Total				total=SUM(total)	⋯	total=sum(Year.total)
7							
8							

```
SELECT Year, Sum(Qnty)
GROUP BY Year
LABEL Sum(Qnty) 'Total Qnty'
```

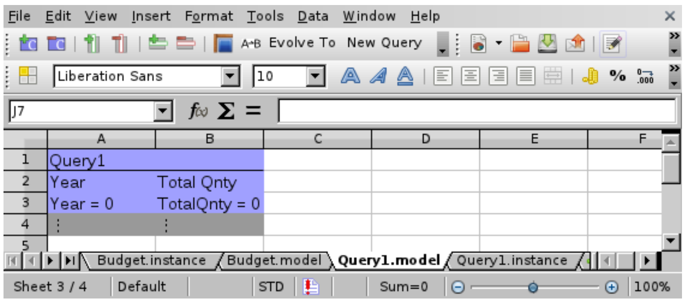
Fig. 7. A model-driven spreadsheet representing Budget information

To demonstrate *QuerySheet*, we will be using the same running ClassSheet model, shown on the left in Figure 7. Suppose we wanted to answer our previous question:

– What was our total quantity per year?

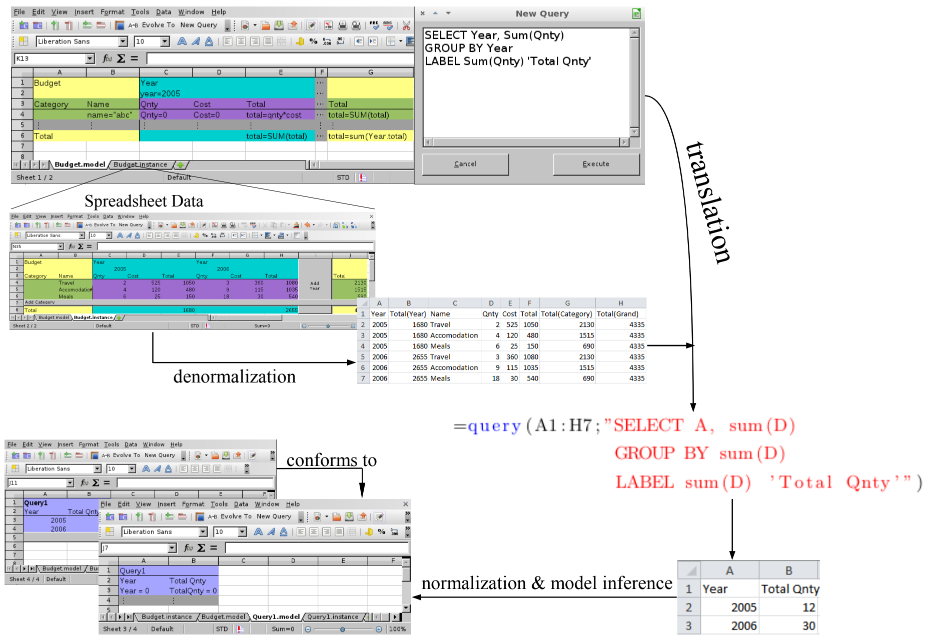
In *QuerySheet*, we can express the query based on the ClassSheet model. The tool provides a *New Query* button, which opens a text box to allow the user to define a query. As we can see in Figure 7 on the right, we have the query for our first question, and as expected, the query looks very much like SQL, using the same keywords and syntactic structure. Moreover, we now use the ClassSheet entities to identify the attributes to be queried.

When executing the query, *QuerySheet* passes through all the phases explained in the previous Sections and shown in Figure 5, while also generating the result as a ClassSheet-driven spreadsheet. In fact, two new worksheets are added to the original spreadsheet: one containing the spreadsheet data that results from the query (Query1.instance), and the other contains the ClassSheet model (Query1.model), as shown in Figure 8. This whole process is depicted in Figure 9.



	A	B
1	Query1	
2	Year	Total Qnty
3	Year = 0	TotalQnty = 0
4	⋮	⋮

Fig. 8. A model-driven spreadsheet inferred from Query1



```
SELECT Year, Sum(Qnty)
GROUP BY Year
LABEL Sum(Qnty) 'Total Qnty'
```

	A	B	C	D	E	F	G	H
1	Year	Total(Year)	Name	Qnty	Cost	Total	Total(Category)	Total(Grand)
2	2005	1680	Travel	2	525	1050	2130	4335
3	2005	1680	Accomodation	4	120	480	1515	4335
4	2005	1680	Meals	6	25	150	690	4335
5	2006	2655	Travel	3	360	1080	2130	4335
6	2006	2655	Accomodation	9	115	1035	1515	4335
7	2006	2655	Meals	18	30	540	690	4335

```
=query(A1:H7;"SELECT A, sum(D)
              GROUP BY sum(D)
              LABEL sum(D) 'Total Qnty'")
```

	A	B
1	Year	Total Qnty
2	2005	12
3	2006	30

Fig. 9. The architecture of *QuerySheet*

7 Empirical Evaluation

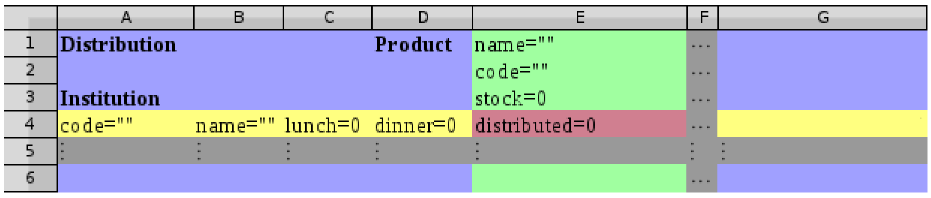
To validate our query system, a study was planned and executed, to obtain results of end-user's experiences, productivity, and feedback. We ran this study one participant at a time. This allowed us to see each participant using our system and learn the difficulties participants were having and how to improve the system to overcome them.

For this study, we had seven students participating, all with basic or minimal knowledge of SQL, who are studying informatics/computer sciences, ranging from Bachelor to PhD students.

For this study we prepared a tutorial to teach them how to use Google's QUERY function and the *QuerySheet* system with a series of exercises using both systems. When the users were comfortable with each system, the actual study was performed.

In the actual study, a real-life spreadsheet was used, which we obtained, with permission to use, from the local food bank in Braga. We then explained to the students how the information was represented, and how to properly read the spreadsheet, in this case, information regarding distributions of basic products and institutions. This specific spreadsheet had information on 85 institutions and 14 different types of basic products, giving way to over 1190 lines of unique information.

We also denormalized the information for the students (since we wanted to study the end-user's interaction with the two different systems, and already knew that denormalizing over 1000 lines of information would take a long time), and also prepared the spreadsheet model and conformed instance in the MDSheet environment. Since we can not show the actual spreadsheet due to revealing private information, only the spreadsheet model (the same one used in the study) is presented below in Figure 10.



	A	B	C	D	E	F	G
1	**Distribution**			**Product**	name=""	...	
2					code=""	...	
3	**Institution**				stock=0	...	
4	code=""	name=""	lunch=0	dinner=0	distributed=0	...	
5	⋮	⋮	⋮	⋮	⋮	⋮ ⋮	
6						...	

Fig. 10. A model-driven spreadsheet representing institutions, products, and distributions, used in the empirical evaluation

As we can see in the model, and hence the actual spreadsheet, the **Distribution** class is composed of a **Institution** class and a **Product** class. The **Institution** class has its **Code** (Institution's Code), **Name** (Intitution's Name), **lunch** (units used for lunch and snacks) and **dinner** (units used for lunch and dinner). The **Product** class has a **Name** (Product's Name), a **Code** (Product's Code), and **Stock** which represents the amount of that specific product they have in stock. The relationship between both classes gives us information on the quantity **Distributed** of a specific **Product** to a specific **Institution**.

For the study, a series of four questions were asked to the students, regarding the information present in the distributions spreadsheet:

1. What is the total distributed for each product?

2. What is the total stock?
3. What are the names of each institution without repetitions?
4. Which were the products with more than 500 units distributed, and which institution were they delivered to?

For each question, they would answer it using Google's QUERY function, and the *QuerySheet* system, alternating between starting with one then the other (the starting system would also alternate between students, so one would begin alternating starting with *QuerySheet*, and another would begin alternating starting with Google's QUERY function). This alternation was introduced in the study so the potential learning from answering a question in one system could not interfere with the results. Since different participants started by answering the same question using different querying systems, the potential learning can be ignored for both systems.

The students were asked to write down the time after carefully reading each question, and the time after the queries were executed with no errors (the correctness of the queries and results were analyzed afterwards), repeating for each system, so they would read the question, write down initial time, write down concluding time, and repeat starting with reading the question once again.

Along with writing down the time, after each question, and having answered it using both systems, the students were asked to choose which system they felt was more: Intuitive, Faster (to write the queries), Easier (to write the queries), Understandable (being able to explain and understand the written queries).

After finishing answering the questions, the students answered which system they preferred and why, and what advantages/disadvantage existed between the systems. Some of the comments can be seen below:

– "*The usage of models helped alot in building the queries. And not having to calculate the range saves time and headaches.*"
– "*Using attribute names instead of column letters is simple and natural.*"
– "*QuerySheet is much more intuitive to use, as simple as looking at the model and attribute names and then I could begin writing queries.*"

The results were gathered and analyzed, and are now presented in Figure 11. The left side (Y-Axis) represents the number of minutes the students took to answer the questions. The bottom side (X-Axis) represents the Question the students answered. The green bars represent the Google QUERY function, and the blue bars represent the *QuerySheet* system.[5]

As we can see, users using the *QuerySheet* system spent significantly less time to write the queries to answer the questions, ranging from as much as 90% less to 40% less, averaging out to 68% faster.

Regarding the system they felt was more Intuitive, Faster, Easier, and Understandable, almost all chose the *QuerySheet* system.

We also analyzed the results and queries written, and in the cases where the queries/results were incorrect, almost all were with the Google QUERY function system, ranging from incorrect column letters chosen, to incorrect ranges.

[5]We assume colors are visible through the digital version of this document.

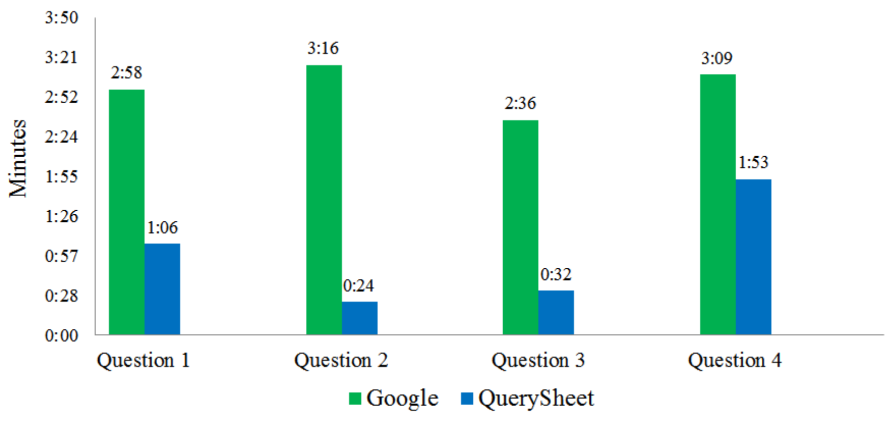


Fig. 11. A chart detailing the information gathered from the empirical evaluation

Furthermore, the written questions at the end also gave us positive feedback. Users stated that using the *QuerySheet* system was much easier to write the query, being able to look at the model to understand the logic behind the information, and not having to deal with calculating the ranges, or worry about positing of information, while being easier to understand what is being written and in turn was more intuitive.

With the user feedback, we were also able to understand what is still needed in *QuerySheet*, such as having a way to store the previous queries for future use. Along with the direct user feedback, we also realized that a basic knowledge of SQL is needed, as expected, to be able to correctly answer the questions. Users who incorrectly wrote queries in the *QuerySheet* system always incorrectly wrote them in Google's QUERY function, due to bad query construction. One of the comments received was to have an interface to build the query visually and not descriptively written, something we already believed would be helpful and needed for a user not used to SQL writing.

8 Conclusion

In this paper, we presented the design and implementation of a query language for model-driven spreadsheets. We designed the query language focusing primarily on how expressive, friendly, readable, and intuitive the queries would be to the users. As our study showed we were able to implement a system that can in fact be used to query spreadsheets in a way users are comfortable with.

Indeed we created a query system that can be used to further knowledge extraction from the spreadsheets. For instance, an interesting way to take advantage of it, is to use it for detecting smells in spreadsheets [21–23], similarly to Fowler's idea of detecting bad smells in source code [24]. With our query language, a user can easily detect a specific bad smell on a spreadsheet, before having to handle possibly critical data. This can even be simplified using a pre-defined set of template queries.

8.1 Future Work

Even with the good results and responses in regards to the work already accomplished, some interesting directions of future research were identified.

Although the empirical results we have presented are interesting, they were the result of a study with only seven participants. We are already planing a second study, this time with more participants so we can confirm our initial results.

Currently, each time a user executes a query, the data is denormalized on-the-spot. A possible way to improve this is to have it so that this full on denormalization is done only once in the beginning, and further changes to data and/or models are changed incrementally, either during the changes, or in the next query execution. An interesting topic which can bring in another level of functionality to the framework, and take advantage of an incremental denormalization, would be synchronization with the query results and original data. By this we mean, allowing a user to, e.g., update the information of one of his/her employees from a previous query result, and in turn this update would reflect upon the original data which the results came from. Acting almost as if the results were a View Table on the original spreadsheet data, possibly using techniques from [25] regarding ways to solve the update-view problems.

References

1. Chambers, C., Scaffidi, C.: Struggling to excel: A field study of challenges faced by spreadsheet users. In Hundhausen, C.D., Pietriga, E., Díaz, P., Rosson, M.B., eds.: VL/HCC, IEEE (2010) 187–194
2. Wadler, P.: Xquery: a typed functional language for querying xml. In: Advanced Functional Programming. Springer (2003) 188–212
3. Belo, O., Cunha, J., Fernandes, J.P., Mendes, J., Pereira, R., Saraiva, J.: Querysheet: A bidirectional query environment for model-driven spreadsheets (tool-demo). In: Proceedings of the 2013 IEEE Symposium on Visual Languages and Human-Centric Computing. VLHCC '13, San Jose, CA, USA, IEEE Computer Society (2013)
4. Maier, D.: The Theory of Relational Databases. Computer Science Press (1983)
5. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer **39**(2) (February 2006) 25–31
6. Bézivin, J.: Model driven engineering: An emerging technical space. In Lämmel, R., Saraiva, J., Visser, J., eds.: GTTSE. Volume 4143 of Lecture Notes in Computer Science., Springer (2005) 36–64
7. Ireson-Paine, J.: Model master: an object-oriented spreadsheet front-end. Computer-Aided Learning using Technology in Economies and Business Education (1997)

8. Abraham, R., Erwig, M., Kollmansberger, S., Seifert, E.: Visual Specifications of Correct Spreadsheets. In: VL/HCC'05: IEEE Symp. on Visual Languages and Human-Centric Computing, IEEE Computer Society (2005) 189–196
9. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: MDSheet: A framework for model-driven spreadsheet engineering. In: Proc. of the 34rd Int. Conf. on Software Engineering, ACM (2012) 1412–1415
10. Mendes, J.: Evolution of model-driven spreadsheets. Master's thesis, University of Minho (2012)
11. Engels, G., Erwig, M.: ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In: Proc. of the 20th IEEE/ACM Int. Conf. on Aut. Sof. Eng., ACM (2005) 124–133
12. Bals, J.C., Christ, F., Engels, G., Erwig, M.: Classsheets - model-based, object-oriented design of spreadsheet applications. In: Proceedings of the TOOLS Europe Conference (TOOLS 2007), Zürich (Swiss). Volume 6., Journal of Object Technology (October 2007) 383–398
13. Mens, T., Demeyer, S., eds.: Software Evolution. Springer (2008)
14. Pereira, R.: Querying for model-driven spreadsheets. Master's thesis, University of Minho (2013)
15. Kankuzi, B., Sajaniemi, J.: An empirical study of spreadsheet authors' mental models in explaining and debugging tasks. In: Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on. (2013) 15–18
16. Melton, J.: Database language sql. In Bernus, P., Mertins, K., Schmidt, G., eds.: Handbook on Architectures of Information Systems. International Handbooks on Information Systems. Springer Berlin Heidelberg (1998) 103–128
17. Shin, S.K., Sanders, G.L.: Denormalization strategies for data retrieval from data warehouses. Decis. Support Syst. **42**(1) (October 2006) 267–282
18. Google: Google query function. `https://developers.google.com/chart/interactive/docs/querylanguage` (2013) [Accessed on November 2013].
19. Google: Google spreadsheet api. `https://developers.google.com/google-apps/spreadsheets` (2013) [Accessed on November 2013].
20. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: IEEE Symp. on Visual Languages and Human-Centric Computing, IEEE CS (2010) 93–100
21. Cunha, J., Fernandes, J.P., Martins, P., Mendes, J., Saraiva, J.: Smellsheet detective: A tool for detecting bad smells in spreadsheets. In Erwig, M., Stapleton, G., Costagliola, G., eds.: VL/HCC, IEEE (2012) 243–244
22. Cunha, J., Fernandes, J.P., Ribeiro, H., Saraiva, J.: Towards a catalog of spreadsheet smells. In: Proceedings of the 12th international conference on Computational Science and Its Applications - Volume Part IV. ICCSA'12, Berlin, Heidelberg, Springer-Verlag (2012) 202–216
23. Hermans, F., Pinzger, M., Deursen, A.v.: Detecting and visualizing inter-worksheet smells in spreadsheets. In: Proceedings of the 2012 International Conference on Software Engineering. ICSE 2012, Piscataway, NJ, USA, IEEE Press (2012) 441–451
24. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
25. Bohannon, A., Vaughan, J.A., Pierce, B.C.: Relational lenses: A language for updateable views. In: Principles of Database Systems (PODS). (2006) Extended version available as University of Pennsylvania technical report MS-CIS-05-27.