Embedding Model-Driven Spreadsheet Queries in Spreadsheet Systems

Jácome Cunha*†, João Paulo Fernandes*‡, Jorge Mendes*, Rui Pereira*, and João Saraiva*
* HASLab/INESC TEC & Universidade do Minho, Portugal
† CIICESI, ESTGF, Instituto Politécnico do Porto, Portugal
‡ RELEASE, Universidade da Beira Interior, Portugal
{jacome,jpaulo,jorgemendes,ruipereira,jas}@di.uminho.pt

***Abstract*—Spreadsheets are widely used not only to define mathematical expressions, but also to store large and complex data. To query such data is usually a difficult task to perform, usually for end user. In this work we embed the textual query language in the model-driven spreadsheet environment as a spreadsheet itself. The result is an expressive and powerful query environment that has knowledge of the business logic defined by the spreadsheet data (the spreadsheet model) to guide end users constructing correct queries.**

I. Introduction

Spreadsheets are widely used and tend to evolve into large and complex data-centric software systems [1]. The use of simple/abstract models to reason about large and complex real-world entities is the standard approach of several engineering disciplines for many decades and even centuries.

More recently, models were widely adopted in software engineering [2], and spreadsheets followed this trend. Indeed, there has been recent and relevant research on model-driven spreadsheets, namely in the definition of spreadsheets models [3], [4], in the inference of models from spreadsheet data [5], [6], in the evolution of spreadsheets [7], [8], [9], and in the construction of model-driven spreadsheet environments [10], [11]. A model-driven spreadsheet developer mimics a civil engineer: instead of reasoning about a large and complex entity/data, he/she reasons about a simpler model defining the business logic the complex data. Synchronization mechanisms are used to guarantee model and instance conformance [12]. A model-driven approach to spreadsheets has furthermore been demonstrated, in different contexts and by different empirical studies, as being both efficient and effective [13].

Surprisingly enough, however, while spreadsheets are such a powerful and multi-purpose system, and namely being used to store large amounts of data, i.e., as databases, spreadsheets still lack the powerful tools and techniques that database systems offer. Specifically, the use of data normalization techniques [14], for data redundancy removal, and a querying language and system, to filter and transform data, is still natively missing in most spreadsheet systems.

In previous work we proposed *QuerySheet*: a textual SQL-like query language and system for model-driven spreadsheets [15]. An empirical study showed that *QuerySheet* greatly improves the productivity of end users when compared to the use of another querying system [16]. Regardless, we also observed that although our technique facilitates querying spreadsheets for users with experience in SQL, those with less experience still felt it difficult to write queries (this was observed in both our system and the competing one), showing that traditional query systems are too complex for end users.

Thus, in this paper we present an Embedded Spreadsheet-Structured Query Language (ES-SQL) that consists in the embedding of an SQL-like query language in a general purpose model-driven spreadsheet system. In such an embedded model-driven spreadsheet query, end users can visually construct their queries as a spreadsheet, and do so in their familiar spreadsheet environment. This is the first contribution of this paper and is shown in Section III. Also, we extend our model/instance synchronization engine to support ES-SQL, thus guaranteeing conformance of all software artifacts after users update/evolve the spreadsheet model or instance. Having the query model/environment synchronized with the spreadsheet model/instance, we exploit this knowledge to guide end users to correctly construct queries: drop down lists are provided containing only possible attributes, aggregations, and ordering types. This is our second contribution and is presented in Section IV.

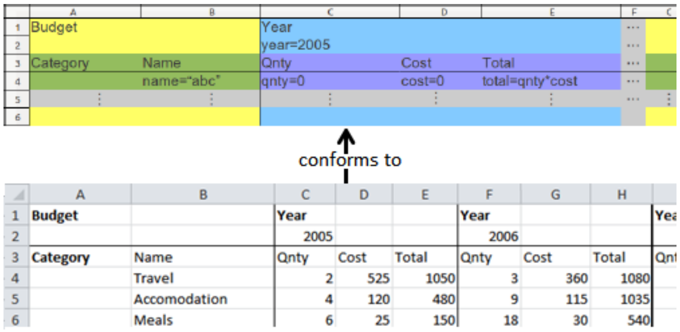
II. Spreadsheets: ClassSheet Models and Queries

When designing and developing our spreadsheet query system, we turned to model-driven engineering methodologies [2]. These methodologies exploit domain specific models, or abstract representations of pieces of software, to handle complex and evolving software systems. We base our work upon model-driven engineering methodologies for spreadsheets, specifically on the spreadsheet modeling language *ClassSheets* [4] and on the model-driven spreadsheet environment *MDSheet* [17].

Models in the *ClassSheet* language are high-level and object-oriented, and use the concepts of classes and attributes. Using these models, we are able to define the business logic of a spreadsheet in a concise and abstract manner. This in turn allows users to understand, evolve, and maintain complex spreadsheets by just analyzing the (*ClassSheet*) models, avoiding looking at large and complex data. Indeed, as shown in [18], users need mental models to build a bridge between spreadsheet data and the real world.

This work is part funded by the 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 project FCOMP-01-0124-FEDER-020484. The first, and fourth author were funded by FCT: SFRH/BPD/73358/2010, BI3-2013_PTDC/EIA-CCO/116796/2010_UMINHO, respectively.

Figure 1 presents our running example: a *ClassSheet* model, and part of a conforming instance, for a spreadsheet containing information of a *Budget*. This *ClassSheet* model, named **Budget**, has a **Category** class (with a **Name** attribute) and a **Year** class (with a **Year** attribute), expanding vertically and horizontally respectively (expressed by the ellipsis). The joining of each class gives us information on the **Quantity**, the **Cost**, and the **Total** of a **Category** in a given **Year**.



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

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

Fig. 1. Budget *ClassSheet* model and conforming instance

Let us suppose we wanted to answer the following question: *What was the total per year, in decreasing order, from 2010 onwards?*

Using our textual model-driven querying approach, we would only need to simply look at our *ClassSheet* model (from Figure 1) and write the query shown in Listing 1 based on our model-driven querying language.

```
SELECT Year, sum(Total)
WHERE Year >= 2010
GROUP BY Year
ORDER BY sum(Total) DESC
```

Listing 1. *QuerySheet* query answering the proposed question

The SQL-like model-driven query is more descriptive than some other approaches, taking advantage of the attribute names used in the model [15], [16]. By doing so, the query itself becomes more human-friendly and understandable to both read and write. Indeed we have shown this in previous work [19].

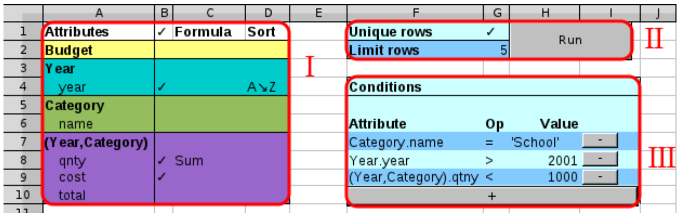
III. Embedding Queries in Model-Driven Spreadsheets

Although constructing queries on a model-driven spreadsheet environment seems to improve efficiency and effectiveness, our empirical observations also suggest that building SQL-like queries may still be too complex for end users. In fact, our studies showed that advanced spreadsheet users with basic SQL experience still felt it difficult to write spreadsheet queries [16]. We observed syntactic and semantic difficulties: first, spreadsheet users were not comfortable to express queries on a textual syntactic SQL-like notation that is very different from the spreadsheet programming paradigm. Second, there is no support on building semantically correct queries, for example, by forcing the query writer to use valid attributes only (i.e., attributes that are in the model).

This section presents the embedded spreadsheet-structure query language that addresses these two issues: First, we define a visual spreadsheet-like query language for model-driven spreadsheets where queries are defined as a spreadsheet. The visual query language is based off of its associated model. Thus, end users write queries in their familiar spreadsheet environment, with no need to learn SQL-like notation. Second, we develop an Integrated Development Environment (IDE) for ES-SQL as an extension of widely used (model-driven) spreadsheet system. This allows an end user to use drop-down boxes to select filter conditions, attribute orders, aggregations, and other querying conditions, to easily construct the queries and eliminating any possible syntax and semantic errors. In other words, this embedded query system helps guide the user in constructing queries.

The focus of the construction of this embedded query language is to display all the information from the model-driven query language in a simple and human-friendly way. This must be intuitive for both experienced SQL users and end users, as suggested by our empirical study. Thus, we define a visual spreadsheet-like representation for each of the syntactic elements in the *QuerySheet* language. Figure 2 shows more advanced features of our embedding that we explain next.

Along with the *ClassSheet* model and conformed instance, the ES-SQL query is also in its own worksheet in the MDSheet model-driven environment.



	A	B	C	D
1	**Attributes**	✓	**Formula**	**Sort**
2	**Budget**			
3	**Year**			
4	year	✓		A↘Z
5	**Category**			
6	name			
7	**(Year,Category)**			
8	qnty	✓	Sum	
9	cost	✓		
10	total			

	F	G	H	I
1	**Unique rows**	✓	Run	
2	**Limit rows**	5		
3				
4	**Conditions**			
5				
6	**Attribute**	**Op**	**Value**	
7	Category.name	=	'School'	-
8	Year.year	>	2001	-
9	(Year,Category).qtny	<	1000	-
10	+			

Fig. 2. ES-SQL: language and spreadsheet IDE

In Figure 2 we see the various areas, identified by the red[1] and Roman numerals. Each area is as follows:

I) Here we have a representation of the spreadsheet model. One can quickly notice that the colors used in this representation come from the original model, allowing the user to familiarize him/herself with the classes in this model representation. This part is divided as follows:

Attributes: shows the class names and associated attributes, based on the original spreadsheet model.

Selected: this column is for the user to select which attributes he/she wishes to be presented in the results (SQL Select Clause). Using drop-down boxes in the cells, the user can choose a *Check Mark* to depict a chosen attribute.

Formula: drop-down boxes in this column present the user with a list of possible formulas to be used on the chosen attribute (Aggregations): Min, Max, Count, Sum, and Avg.

Sort: to allow the user to sort the results using a specific attribute (SQL Order By Clause), a series of drop-down boxes in the cells presents the user the option of an Ascendant (Z↘A) or Descendant (A↘Z) sorting.

II) Here we have two operations to be applied over the rows of the results of a query, and a "Run" button. This is divided as follows:

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

Unique Rows: this is used to produce distinct results (Distinct clause), removing any repetitions. Using the drop-down box (under column G), the user can choose a *Check Mark* to express the wish to produce unique rows of information.

Limit Rows: this is to limit the number of rows to be shown in the results. The user can write a number in the cell (under column G) to state how many rows to be shown. For example, Limit rows 5 would only show the first 5 rows of information from the query result.

"Run" Button: when the user is done constructing the desired query, he/she may click on the "Run" button to execute the query and produce the results.

III) Displays the conditions to be used in the query. Clicking on the "+" button adds a new row; clicking "-" removes the corresponding row. This is divided as follows:

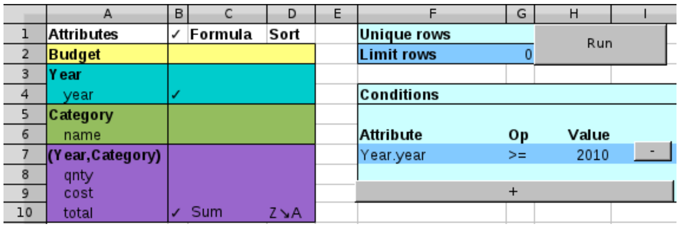
Attribute: the user is presented a drop-down box allowing to choose which attribute to be used for a condition.

Op: a second drop-down box is presented, allowing the user to choose which operation to be used for a condition. The operations are: =, <, >, <=, >=, !=.

Value: here a user may write in the actual cell, the value to be compared to the attribute with the operation.

Another useful addition is the automatic calculation of when a group by is needed. In other words, when an aggregation is detect with other selected attributes, the embedded query automatically produces a grouping. This automatic calculation not only is practical in query construction, but also made it so one less query clause was needed to be presented.

Having introduced ES-SQL, we can now construct the query from Listing 1 using our approach, as shown in Figure 3.



	A	B	C	D	E	F	G	H	I
1	Attributes	✓	Formula	Sort		Unique rows		Run	
2	Budget					Limit rows	0		
3	Year								
4	year	✓				Conditions			
5	Category								
6	name					Attribute	Op	Value	
7	(Year,Category)					Year.year	>=	2010	-
8	qnty								
9	cost						+		
10	total	✓	Sum	Z↘A					

Fig. 3. Embedded Query answering running example

The steps to construct this embedded query are as follows:

1) Using the drop-down boxes under the *Selected* column, click on cell B4 and B10 to select the *Check Mark*, selecting both *year* and *total* to be used, respectively.
2) Under the *Formula* column, click on cell C10 and choose *Sum* from the drop-down box list.
3) Click on the "+" button to add a new condition row.
4) Select the *Year.year* attribute, and >= operation using the drop-down boxes in the new condition row. Afterwards, fill in *2010* in the value cell.
5) Click "Run".

As we have shown, using ES-SQL, the user can have little to no SQL experience, and still correctly perform queries.

IV. Spreadsheet Models, Instances and Queries Synchronization

In the previous section we have presented the embedding of model-driven visual queries. Such query system is devised from the spreadsheet model, the ClassSheet, and must always be synchronized with such a model so the queries are always correct, even after model evolution. Thus, this section presents the environment we have designed to keep these artifacts synchronized. Indeed we extend our model-driven spreadsheet bidirectional evolution engine presented in [7], [8], [12] to allow model, instance, and query synchronization. In such a setting, the embedded query model has knowledge of the underlying spreadsheet model (and instance), and automatically (co)evolves after the model/instance evolves. Because the query model is synchronized with the spreadsheet model, it offers to users information about valid and available attributes currently in the spreadsheet so that it guides in constructing correct queries. Figure 4 illustrates this system.

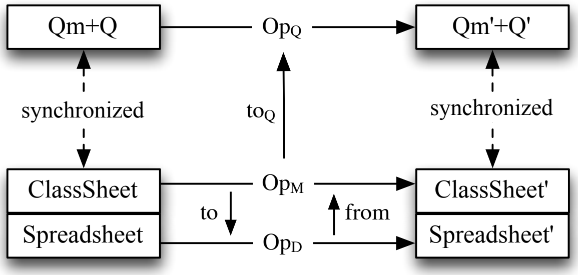


Fig. 4. Diagram of the synchronization between model, data, and queries.

The transformation engine, with the extension presented here, operates on three distinct structures: the *Model* (ClassSheet), *Data* (Spreadsheet), and *Query* (Qm+Q). The interaction between model and instance was presented in [20], and is presented here for completeness purposes: For each evolution step performed on the model, Op_M, a corresponding operation on the data is calculated, Op_D. This is done by the function *to*. On the other hand, for every operation on the instance, *from* calculates the corresponding operation it is necessary to perform on the model to keep both artifacts in conformance. This can be seen in the bottom part of the diagram presented in Figure 4. With the work presented in this paper we will complete the top part of this diagram. Thus, we have the query model and each query instance ($Qm+Q$) synchronized with the model. The first query model is devised from the ClassSheet, but if the model change, the query must automatically be updated. To do so, for every update of the model, Op_M, the function to_Q will compute the necessary changes to the corresponding query model and possible existing queries.

V. Related Work

In the past, we have proposed the first model-driven approach to spreadsheet querying [15], [16]. This initial proposal includes a language and a tool to construct queries over spreadsheets with models. In contrast with the work we present in this paper, such approach was not well suited for end users: to construct queries users would need to already know or go trough a steep learning path of SQL. In fact, we have designed and conducted some empirical studies which showed that even users with computer science background would have some difficulties constructing such queries [16]. In an attempt to overcome this issue, we have proposed a more graphical approach to help end users perform queries over spreadsheets [19]. With this approach we were able to show that users were now more capable of writing queries [19]. Moreover, we have also implemented the embedded querying system under MDSheet [21].

Microsoft has its own tool to query Excel files: *MS-Query* [22]. This is actually a database query interface used by Microsoft Word and Excel, and consists of an utility which imports databases, text files, and other spreadsheet representations (such as csv) into Excel. After importing, a query builder wizard is shown to begin constructing a query. In order to be able to query spreadsheet data, the headers of each attribute must be explicitly represented in a single row and the data itself must be in a single vertically expanding table format, format that does not exist for many spreadsheets.

Another tool which allows to query spreadsheets is the Google QUERY function [23]. It allows users, using a SQL-like syntax, to perform a query over an array of values, for example in their Google Drive spreadsheets, where the function is built-in. Google QUERY function is a two part query, consisting of a range of input cells, and the actual query string (subset of SQL), referring data by the names of the columns where it is placed. The QUERY function shares the same problems of *MS-Query* in regards to data representation.

Open/LibreOffice have also a way of querying spreadsheets [24]. In this case, it must be done through the database component of the software, and not in the spreadsheet system itself. The user must create a new database, choosing an existing spreadsheet file as its source.

VI. Conclusions and Future work

In this paper, we have proposed a model-driven querying artifact. Spreadsheets are abstractly reasoned about, and evolved, together with a ClassSheet model representation of them. The fact that a model which captures the business logic of a spreadsheet is available is exploited in that queries may intuitively refer to names of entities instead of column headers as in previous solutions.

Our approach builds on the language for defining queries that we propose. Such a language also relies on the model available: A query template is offered to users, that construct a query by selecting the parts of information they are interested in; this is done without margin to mistakes, as the template only makes available the choices that are actually possible.

Finally, the fact that our query engine is now fully embedded in a spreadsheet system offers: i) an improved user experience, with queries readily available under spreadsheet systems and ii) interesting possibilities for full synchronization of all the elements that are manipulated: ClassSheet models, spreadsheets, queries and query results. This means upon an evolution of any of these elements, we now have the means to synchronize all the associated ones.

The work so far opens interesting research problems. While able to represent most of our model-driven domain-specific query in spreadsheets, to write nested logical conditions is still a limitation. Also, although our previous experience with querying spreadsheets seem to confirm the interest of this work, we still need to validate this claim empirically.

References

[1] C. Chambers and C. Scaffidi, “Struggling to Excel: A Field Study of Challenges Faced by Spreadsheet Users,” in *VL/HCC’10*, 2010, pp. 187–194.

[2] J. Bézivin, “Model driven engineering: An emerging technical space,” in *GTTSE’05*, ser. LNCS, R. Lämmel, J. Saraiva, and J. Visser, Eds., vol. 4143. Springer, 2005, pp. 36–64.

[3] R. Abraham and M. Erwig, “Inferring templates from spreadsheets,” in *ICSE’06*. ACM, 2006, pp. 182–191.

[4] G. Engels and M. Erwig, “ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications,” in *ASE’05*. ACM, 2005, pp. 124–133.

[5] F. Hermans, M. Pinzger, and A. van Deursen, “Automatically extracting class diagrams from spreadsheets,” in *ECOOP’10*. Springer-Verlag, 2010, pp. 52–75.

[6] J. Cunha, M. Erwig, and J. Saraiva, “Automatically Inferring ClassSheet Models from Spreadsheets,” in *VL/HCC’10*. IEEE, 2010, pp. 93–100.

[7] J. Cunha, J. Visser, T. Alves, and J. Saraiva, “Type-safe evolution of spreadsheets,” in *FASE’11*. Springer-Verlag, 2011, pp. 186–201.

[8] J. Cunha, J. Mendes, J. P. Fernandes, and J. Saraiva, “Embedding and evolution of spreadsheet models in spreadsheet systems,” in *VL/HCC’11*. IEEE, 2011, pp. 186–201.

[9] M. Luckey, M. Erwig, and G. Engels, “Systematic evolution of model-based spreadsheet applications,” *Journal of Visual Languages & Computing*, vol. 23, no. 5, pp. 267–286, 2012.

[10] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein, “Gencel: a program generator for correct spreadsheets,” *J. Funct. Program*, vol. 16, no. 3, pp. 293–325, 2006.

[11] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, “MDSheet: A framework for model-driven spreadsheet engineering,” in *ICSE’12*. IEEE Press, 2012, pp. 1395–1398.

[12] J. Cunha, J. P. Fernandes, J. Mendes, H. Pacheco, and J. Saraiva, “Bidirectional transformation of model-driven spreadsheets,” in *ICMT’12*. Springer-Verlag, 2012, pp. 105–120.

[13] L. Beckwith, J. Cunha, J. P. Fernandes, and J. Saraiva, “End-users productivity in model-based spreadsheets: An empirical study,” in *IS-EUD’11*, ser. LNCS. Springer Berlin Heidelberg, 2011, pp. 282–288.

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

[15] J. Cunha, J. P. Fernandes, J. Mendes, R. Pereira, and J. Saraiva, “Querying model-driven spreadsheets,” in *VL/HCC’13*. IEEE, 2013, pp. 83–86.

[16] J. Cunha, J. Mendes, J. P. Fernandes, R. Pereira, and J. Saraiva, “Design and implementation of queries for model-driven spreadsheets,” in *Proceedings of the DSL Summer School 2013*, 2014, (submitted).

[17] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, “MDSheet: A framework for model-driven spreadsheet engineering,” in *ICSE’12*. IEEE Press, 2012, pp. 1412–1415.

[18] B. Kankuzi and J. Sajaniemi, “An empirical study of spreadsheet authors’ mental models in explaining and debugging tasks,” in *VL/HCC’13*. IEEE, 2013, pp. 15–18.

[19] J. Cunha, J. P. Fernandes, R. Pereira, and J. Saraiva, “Graphical querying of model-driven spreadsheets,” in *HCII’14*, ser. LNCS. Springer, 2014.

[20] J. Cunha, J. P. Fernandes, J. Mendes, H. Pacheco, and J. Saraiva, “Bidirectional transformation of model-driven spreadsheets,” in *ICMT’12*, ser. LNCS, vol. 7307. Springer, 2012, pp. 105–120.

[21] J. Cunha, J. P. Fernandes, J. Mendes, R. Pereira, and J. Saraiva, “ES-SQL: Visually Querying Spreadsheets,” in *VL/HCC’14*. IEEE, 2014, to appear.

[22] Microsoft Query, office.microsoft.com/en-us/excel-help/use-microsoft-query-to-retrieve-external-data-HA010099664.aspx.

[23] Google Query, developers.google.com/chart/interactive/docs/querylanguage.

[24] LibreOffice Query, help.libreoffice.org/Common/Query_Design.