Querying Model-Driven Spreadsheets

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 being used with many different purposes that range from toy applications to complete information systems. In any of these cases, they are often used as data repositories that can grow significantly. As the amount of data grows, it also becomes more difficult to extract concrete information out of them.**

This paper focuses on the problem of spreadsheet querying. In particular, we propose an expressive and composable technique where intuitive queries can be defined. Our approach builds on a model-driven spreadsheet development environment, and queries are expressed referencing entities in the model of a spreadsheet instead of in its actual data. Finally, the system that we have implemented relies on Google's query function for spreadsheets.

I. Introduction

Spreadsheet systems have proven to be a true success story, both in the context of the development of simple *applications* for personal use as well as to support business decisions within large industrial organizations.

A significant contribution to the success of spreadsheets is acknowledged to their multipurposeness which can be observed, for example, by the fact that many spreadsheets are actually used as data repositories or databases [1], in the sense that no formulas are defined in them[1].

From a computer scientist's point-of-view, however, it is clear that spreadsheets do not support the state-of-the-art techniques that are incorporated in modern database management systems and so their use as databases raises a number of issues. One such issue is that data in spreadsheets often does not conform to normalization principles. Indeed, and just as a simple evidence of this, the same data in a spreadsheet (*e.g.*, an address) is often repeated in many cells of a spreadsheet, and an update in one of those cells that is not propagated to all the others results in data inconsistency (the same address will then have multiple *definitions*). The problem of normalization of spreadsheets has already been addressed in [2], and this is actually a technique that we exploit in the context of this work.

In this paper, we address another significant issue that arises from the use of spreadsheets as databases: the problem of data querying. In fact, having all the data stored in a single and potentially large matrix makes it difficult to extract information from such data in an efficient and effective way. Again, the problem of data querying is one that has received decades of attention within the database community but only recently has been considered in the context of spreadsheets. This first approach to spreadsheet querying was proposed by Google [3] and has already been integrated within Google Drive. While in our context we also exploit Google's approach to querying, it can still be improved.

For once, Google's method limits the way the data must be represented, reducing the freedom of how one would want to represent data: it requires data to be in a tabular format, with headers present in the first row, and does not support relationships between data entities, requiring the data to be denormalized if one would like to take advantage of all the different query possibilities.

Along with the difficulty of managing and structuring the data in such a way, the proposed method has another flaw - the actual SQL-like query. Instead of writing the query using column labels, one must use the column letters to write the query, leading to counter-intuitive query writing, and hard to understand queries, for both the user who wrote the query and for anyone trying to read and understand it. And since spreadsheets tend to include large amounts of data (columns and rows), it becomes extremely difficult to understand the business logic of the data, and know the exact columns we need to refer to in the query.

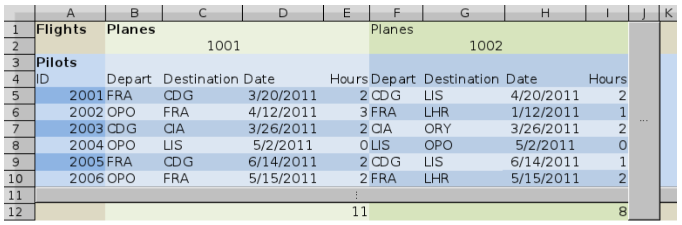
In this paper, we propose to lift spreadsheet querying to the model level. Our approach envisions a spreadsheet development setting where a spreadsheet holding concrete data coexists with an abstract model of its structure. This is a setting that has already been realized and implemented before [4], providing a bidirectional model-driven spreadsheet development environment within a spreadsheet system itself [5], [6], where both models and data instances are automatically kept always synchronized, even when one of these artefacts evolves.

In such a domain, we impose no limits on the structure that data needs to conform to, overcoming the previous approach's first limitation. Also, we allow queries to be expressed by names which reference structure entities in the model, therefore improving the readability and maintainability of queries, and overcoming the second limitation described before. We argue that our approach provides a powerful and expressive information extraction querying setting where queries are furthermore simple to write, read, understand and maintain.

II. Querying Spreadsheets

Before we discuss techniques to query spreadsheets, let us introduce a spreadsheet that will be used as a running example throughout this document. Figure 1 presents a spreadsheet for storing information about flights, containing the information

[1]While, of course, this suggests that many spreadsheets should be converted to databases, in practice we do not observe this migration often.



	A	B	C	D	E	F	G	H	I	J	K
1	**Flights**	**Planes**				Planes					
2			1001				1002				
3	**Pilots**										
4	ID	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours		
5	2001	FRA	CDG	3/20/2011	2	CDG	LIS	4/20/2011	2		
6	2002	OPO	FRA	4/12/2011	3	FRA	LHR	1/12/2011	1		
7	2003	CDG	CIA	3/26/2011	2	CIA	ORY	3/26/2011	2	...	
8	2004	OPO	LIS	5/2/2011	0	LIS	OPO	5/2/2011	0		
9	2005	FRA	CDG	6/14/2011	2	CDG	LIS	6/14/2011	1		
10	2006	OPO	FRA	5/15/2011	2	FRA	LHR	5/15/2011	2		
11					⋮						
12					11				8		

Fig. 1. A spreadsheet representing flight information.

about the pilot, the plane, the departure and destination cities, date and duration of the flight.

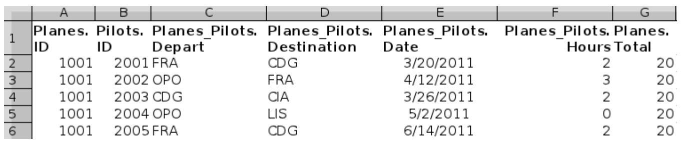
Having the flight information available, a flight manager may want to know, for example, which flights have their destination as Lisbon (`LIS`), or how many hours a given pilot has flown. In standard database systems such information can be easily computed by defining simple SQL queries. However, using traditional spreadsheet systems it is not easy/possible to compute such information.

A. Examples of Query Systems

Having realized the need to provide queries for spreadsheets, both Microsoft and Google made an attempt to allow queries to be used on spreadsheets, with their *MS-Query Tool* and *Google query function* (GQF) respectively. In this section we briefly present the drawbacks with these two attempts.

One issue both share, is the necessary representation of the data to be able to run the queries. Both require spreadsheet data to be represented in a tabular format, with headers present in the first row. They do not support relationships between data entities, or in other words, they require the data to be denormalized [7], [8].

For instance, the data displayed on Figure 1 cannot be completely queried. The planes information would need to appear in front of each flight. The denormalized data is shown in Figure 2. This format is hard to understand, making even more complex to write spreadsheet queries. This denormalization process is the opposite of recommendations from the database realm because it creates *update* and *delete anomalies* [7]. This subject will be further discussed in sub-section III-B.



	A	B	C	D	E	F	G
1	**Planes. ID**	**Pilots. ID**	**Planes_Pilots. Depart**	**Planes_Pilots. Destination**	**Planes_Pilots. Date**	**Planes_Pilots. Hours**	**Planes. Total**
2	1001	2001	FRA	CDG	3/20/2011	2	20
3	1001	2002	OPO	FRA	4/12/2011	3	20
4	1001	2003	CDG	CIA	3/26/2011	2	20
5	1001	2004	OPO	LIS	5/2/2011	0	20
6	1001	2005	FRA	CDG	6/14/2011	2	20

Fig. 2. Part of the flight spreadsheet data in a denormalized state.

Along with the aforementioned drawback, the Google query function has another problem. A quick glance over Listing 1 shows us that the query consists of a *range* input, to state the range of cells where the data to be queried is present, and the actual *query string*, which is a subset of SQL where column letters are used instead of attribute names.

```
=query(A1:G53;
  "SELECT B, SUM(F)
  WHERE D = 'LIS'
  GROUP BY B
  LABEL SUM(F) 'Total Hours'")
```

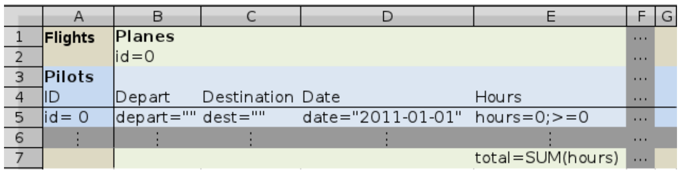
Listing 1. A Google query that counts the number of hours each pilot flew with their destination as Lisbon.

Since Google queries use column letters to define the spreadsheet data area/matrix and the columns/attributes needed in expressing the query, they lead to counter-intuitive and hard to understand queries, specially in large and complex spreadsheets. In fact, the user needs to know the column letters where the information referred in the query is stored in the spreadsheet. Moreover, Google queries do not adapt/evolve when the spreadsheet data evolves. That is to say, by adding a column to the spreadsheet we may turn a query invalid, because data changed places.

Yet even with these drawbacks, Google queries can be written in the spreadsheet itself, as regular content of cells, allowing on-the-spot results without any configurations, and providing a powerful query engine for spreadsheets. For these reasons and because it is a free service, we chose to incorporate the Google query function in our work and take advantage of this query engine.

B. Model-Driven Spreadsheets

Because spreadsheets may evolve into complex software systems, Engels and Erwig [9] introduced ClassSheets as a model formalism to express the business logic spreadsheet data. The ClassSheet formalism provides a model-driven software development approach to spreadsheets. Thus, the business logic of the spreadsheet is defined in an abstract and concise formalism. As a result, users can understand, maintain and evolve complex spreadsheets by just analysing their (ClassSheet) models, and not by looking at large and complex data. Figure 3 defines a ClassSheet for our flight spreadsheet.



	A	B	C	D	E	F	G
1	**Flights**	**Planes**				...	
2		id=0				...	
3	**Pilots**					...	
4	ID	Depart	Destination	Date	Hours	...	
5	id= 0	depart=""	dest=""	date="2011-01-01"	hours=0;>=0	...	
6	⋮	⋮	⋮	⋮	⋮	...	
7					total=SUM(hours)	...	

Fig. 3. ClassSheet model for the flights' spreadsheet presented in Figure 1.

ClassSheet is a high-level and object-oriented formalism. It uses the notion of class and attribute. In our example, **Flight** is a class that is composed of both a **Pilot** and a **Plane** class, expanding vertically and horizontally, respectively, and each with its own identification code, or **ID**. The joining of the **Pilot** and **Plane** classes gives us four distinct attributes: **Depart**, **Destination**, **Date**, and **Hours**, each with its own default value.

This ClassSheet specifies the business logic of our flight spreadsheet. In model-driven engineering we say that the spreadsheet data (Figure 1) *conforms to* the model (Figure 3).

Having defined the model of our spreadsheet flight example, we can now define the query based only on the model. The query can reference attributes instead of column letters, as in GQF. Compared with Google queries, this is probably easier to manage. Moreover, it makes the query robust to evolution: if new columns are added, there is no need to change the queries. Indeed, no denormalization process is necessary to query spreadsheets: only a model is necessary. For those spreadsheets that do not have a model, the work presented in [10] can be used to automatically infer one. The query presented before can now be written as in Listing 2.

```
SELECT Pilots.*, Sum(Hours)
FROM Flights
WHERE Destination = 'LIS'
GROUP BY Pilots.*
LABEL Sum(Hours) 'Total Hours'
```

Listing 2. Proposed SQL-like query for the original query.

III. Querying Model-Driven Spreadsheets

In this section we explain how the query system we have created works. Figure 4 presents its architecture. This system in implemented on top of MDSheet [4]. This means that all the mechanisms to handle models and instances are already created and ready to use. This is the 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 sub-section. The spreadsheet instance is then denormalized, as we will explain in sub-section III-B, and the query over the model is translated into a Google query, as explained in sub-section III-C. 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. Finally, a new model is inferred so the result can be used as input to a new query, as explained in sub-section III-D. This last step is necessary since we want the queries to be composable.

A. *Model-Driven Query Language*

The Model-Driven Query Language is based much off of the standard SQL language, while allowing some of the GQF's clauses such as LIMIT and LABEL. Its syntax is shown in Listing 3. Instead of selecting table attributes in the SELECT clause, the user selects the ClassSheet attributes he/she wishes to query, while allowing him/her to specify, to avoid conflicts, various ways of naming the attribute, such as simply stating its name (**Destination**), with the class in which the attribute is present (**Pilots.ID**), with both classes, in the case of a relational class (**(Pilots,Planes).Destination**), or if one wants all the attributes in a certain class, only state the class's name (**Pilots**).

```
SELECT (* | attr1, attr2, ...)
FROM ClassSheet1, [JOIN ClassSheet2 ON attr], ...
[WHERE conditions]
[GROUP BY attr1, ...]
[ORDER BY attr1 [ASC|DESC], ...]
[LIMIT numRow]
[LABEL attr1 new_attr1, ...]

attr ::= attribute
    | Class
    | Class.attribute
    | (Class1, Class2).attribute
```

Listing 3. The model-driven query language syntax.

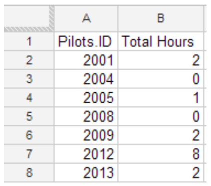
The FROM clause allows the user to choose which ClassSheet model(s) to use for the query, reminiscent of the standard SQL FROM clause where the user chooses what tables to query. Notice that as in SQL we allow JOIN operations. Basing this language off of traditional SQL allows users who already know 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.

B. *Denormalization of Spreadsheet Data*

As we stated before, the GQF needs to receive all the data in a single table. Unfortunately, this has some disadvantages like data redundancy that can possibly lead to data inconsistency. In fact, these are well-known and well-studied problems in the database realm [7]. Thus, for our system to be able to use the Google query system, we need to map all data to such format. In previous work we created a technique to migrate data from spreadsheets to a relational database and back [2]. This technique receives a spreadsheet in the same format required by the GQF and normalizes it. The technique we presented is bidirectional, meaning that it is possible to go from normalized data to a denormalized format, as we need. Thus, this part of our framework is handled by the same mechanism presented in [2].

C. *Translation to Google Query*

To correctly run the GQF, the query must abide by the *Visualization API Query Language* [3], defined by Google. For our model-driven queries to function properly, a translator was made, with the sole purpose to transform the model-driven query written by the user into its exact counterpart for the GQF. It also automatically calculates the *range* of the ClassSheet models selected in the FROM clause, and substitutes the attribute names to their corresponding column letters, without the user having to do so. The translator automatically writes the *range* of the ClassSheet models selected in the FROM clause, by using a quick lookup function to find what is the *range* of the selected class. After verifying that each attribute chosen by the user exists, and has no conflicts, such as ambiguous attribute names due to the attribute repeating in more than one ClassSheet (which may be solved by adding the class name beforehand as shown in III-A), another lookup function is used for each attribute which returns the exact column letter corresponding to the attribute. Having both the translated model-driven query and denormalized data, we can now use the GQF, taking advantage of this powerful tool, and obtain the desired results as shown in Figure 5.

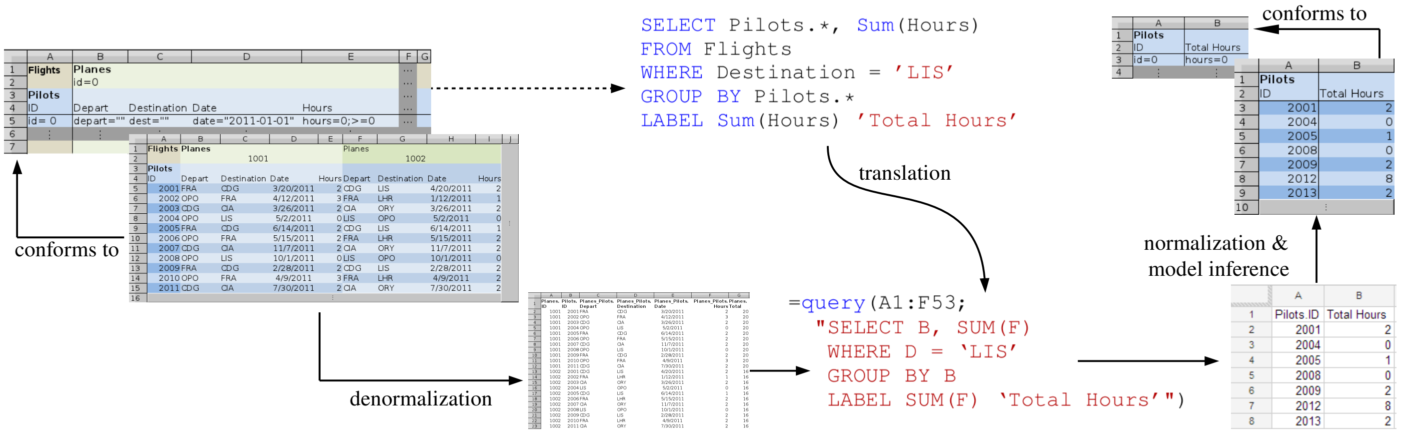


	A	B
1	Pilots.ID	Total Hours
2	2001	2
3	2004	0
4	2005	1
5	2008	0
6	2009	2
7	2012	8
8	2013	2

Fig. 5. Data resulting from a Google query.

D. *Normalization and ClassSheet Inference*

After applying a Google query, its result is shown in the form of a spreadsheet table. For instance, if we perform the



	A	B	C	D	E	F	G
1	**Flights**	**Planes**				...	
2		id=0				...	
3	**Pilots**					...	
4	ID	Depart	Destination	Date	Hours	...	
5	id= 0	depart=""	dest=""	date="2011-01-01"	hours=0;>=0	...	
6	⋮	⋮					
7							

	A	B	C	D	E	F	G	H	I
1	**Flights**	**Planes**				Planes			
2			1001				1002		
3	**Pilots**								
4	ID	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours
5	2001	FRA	CDG	3/20/2011	2	CDG	LIS	4/20/2011	2
6	2002	OPO	FRA	4/12/2011	3	FRA	LHR	1/12/2011	1
7	2003	CDG	CIA	3/26/2011	2	CIA	ORY	3/26/2011	2
8	2004	OPO	LIS	5/2/2011	0	LIS	OPO	5/2/2011	0
9	2005	FRA	CDG	6/14/2011	2	CDG	LIS	6/14/2011	1
10	2006	OPO	FRA	5/15/2011	2	FRA	LHR	5/15/2011	2
11	2007	CDG	CIA	11/7/2011	2	CIA	ORY	11/7/2011	2
12	2008	OPO	LIS	10/1/2011	0	LIS	OPO	10/1/2011	0
13	2009	FRA	CDG	2/28/2011	2	CDG	LIS	2/28/2011	2
14	2010	OPO	FRA	4/9/2011	3	FRA	LHR	4/9/2011	2
15	2011	CDG	CIA	7/30/2011	2	CIA	ORY	7/30/2011	2
16					⋮				

```
SELECT Pilots.*, Sum(Hours)
FROM Flights
WHERE Destination = 'LIS'
GROUP BY Pilots.*
LABEL Sum(Hours) 'Total Hours'
```

```
=query(A1:F53;
  "SELECT B, SUM(F)
   WHERE D = 'LIS'
   GROUP BY B
   LABEL SUM(F) 'Total Hours'")
```

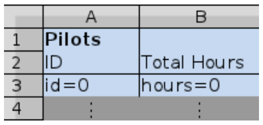
	A	B
1	Pilots.ID	Total Hours
2	2001	2
3	2004	0
4	2005	1
5	2008	0
6	2009	2
7	2012	8
8	2013	2

	A	B
1	**Pilots**	
2	ID	Total Hours
3	2001	2
4	2004	0
5	2005	1
6	2008	0
7	2009	2
8	2012	8
9	2013	2
10	⋮	

	A	B
1	**Pilots**	
2	ID	Total Hours
3	id=0	hours=0
4	⋮	⋮

Fig. 4. Architecture of the model-driven query system.

query we have been using in this paper, we obtain the data shown in Figure 5. One of the goals of our framework is to make queries composable, that is, we want to use the output of a query as input of another query (as indeed happens with Google query). Without a model for the output of the Google query this is not possible. In previous work we have introduced a technique to automatically infer a ClassSheet from spreadsheet data [10]. Thus, such a technique can directly be applied to fulfill this gap in our framework. Applying the inference technique to the spreadsheet presented in Figure 5 we obtain the ClassSheet shown in Figure 6.



	A	B
1	**Pilots**	
2	ID	Total Hours
3	id=0	hours=0
4	⋮	⋮

Fig. 6. Model automatically inferred from the spreadsheet shown in Figure 5.

IV. Related Work and Conclusion

In this paper, we have proposed a framework for the querying of spreadsheets that are developed within a model-driven environment. The setting that we provide is highly expressive, and its queries are intuitive, easy to write, read and understand. Querying is an important aspect of database theory and systems. Recently, query languages have been adopted in other settings, like XML (with XQuery [11]), spreadsheets and general-purpose programming languages [12]. Google embedded SQL in Google Docs spreadsheets and Microsoft defined a database query interface used by Microsoft Word and Excel.

Several aspects of our work deserve further elaboration. In particular, after performing a query we wish to derive models that are as close as possible to the original ones. We also plan to execute empirical studies with regards to the efficiency of the query system and how expressive, readable, and intuitive the queries are to users.

Acknowledgment

This work is 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-020532. This work is financed 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-022701. The four first authors were funded by FCT: SFRH/BPD/73358/2010, SFRH/BPD/46987/2008, BI3-2012PTDC/EIA-CCO/108613/2008, BI2-2012PTDC/EIA-CCO/1086613/2008 respectively.

References

[1] C. Chambers and C. Scaffidi, "Struggling to excel: A field study of challenges faced by spreadsheet users," in *VL/HCC*, C. D. Hundhausen, E. Pietriga, P. Díaz, and M. B. Rosson, Eds. IEEE, 2010, pp. 187–194.

[2] J. Cunha, J. Saraiva, and J. Visser, "From spreadsheets to relational databases and back," in *PEPM'09: Proc. of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program manipulation*. ACM, 2009, pp. 179–188.

[3] Google, "Google query function," http://goo.gl/p0FKW, [Accessed on March 2013].

[4] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, "MDSheet: A framework for model-driven spreadsheet engineering," in *Proc. of the 34rd Int. Conf. on Software Engineering*. ACM, 2012, pp. 1412–1415.

[5] J. Cunha, J. Mendes, J. P. Fernandes, and J. Saraiva, "Embedding and evolution of spreadsheet models in spreadsheet systems," in *IEEE Symp. on Visual Lang. and Human-Centric Comp.* IEEE, 2011, pp. 186–201.

[6] 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.

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

[8] S. K. Shin and G. L. Sanders, "Denormalization strategies for data retrieval from data warehouses," *Decis. Support Syst.*, vol. 42, no. 1, pp. 267–282, Oct. 2006.

[9] G. Engels and M. Erwig, "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, pp. 124–133.

[10] J. Cunha, M. Erwig, and J. Saraiva, "Automatically inferring classsheet models from spreadsheets," in *VL/HCC'10: IEEE Symp. on Visual Languages and Human-Centric Computing*. IEEE Computer Society, 2010, pp. 93–100.

[11] D. Chamberlin, “Xquery: An xml query language,” *IBM Syst. J.*, vol. 41, no. 4, pp. 597–615, Oct. 2002.

[12] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble, “.ql: Object-oriented queries made easy,” in *GTTSE*, ser. Lecture Notes in Computer Science, R. Lämmel, J. Visser, and J. Saraiva, Eds., vol. 5235. Springer, 2007, pp. 78–133.