MDSheet: A Framework for Model-Driven Spreadsheet Engineering

Jácome Cunha*, João Paulo Fernandes*† Jorge Mendes*, João Saraiva*
*HASLab / INESC TEC, Universidade do Minho, Portugal
†Departamento de Engenharia Informática, Universidade do Porto, Portugal
{jacome,jpaulo,jorgemendes,jas}@di.uminho.pt

***Abstract*—In this paper, we present MDSHEET, a framework for the embedding, evolution and inference of spreadsheet models. This framework offers a model-driven software development mechanism for spreadsheet users.**

***Keywords*-Spreadsheets; Model-Driven Engineering (MDE); Software Evolution; Embedded DSLs; Model Inference**

I. INTRODUCTION

Spreadsheets are widely used by non-professional programmers, the so-called *end users*, to develop business applications. Spreadsheet systems offer end users a high level of flexibility, making it easier to get started working with them. This freedom, however, comes with a price: spreadsheets are error prone as shown by several industrial case studies reporting that an average of 94% of the analyzed spreadsheets contained errors [1].

As programming systems, spreadsheets lack the support provided by modern programming languages/environments, like for example, higher-level abstractions and powerful type and modular systems. As a result, they are prone to errors. In order to improve end-users productivity, several techniques have been proposed, which guide end users to safely and correctly edit spreadsheets, like, for example, the use of spreadsheet templates [2], *ClassSheets* [3], [4], and the inclusion of visual objects to provide editing assistance in spreadsheets. All these approaches propose a form of end user model-driven software development: a spreadsheet business model is defined, from which a customized spreadsheet application is then generated guaranteeing the consistency of the spreadsheet data with the underlying model.

In this paper, we present MDSHEET, a unifying framework where we have integrated the following modelling, manipulation and co-evolution spreadsheet techniques:

- Embedding of *ClassSheet* models: *ClassSheets* are a powerful and widely used modelling language to define the business logic of a spreadsheet. MDSHEET embeds this modelling language in a spreadsheet system, providing a coherent environment for model driven spreadsheet engineering, as proposed in [5].
- Co-evolution of *ClassSheets* and instances: Like any other software artifact, spreadsheets evolve over time. MDSHEET uses a formal setting where the co-evolution of the embedded *ClassSheet* model and the spreadsheet instance is performed [6].
- Inference of *ClassSheets*: Being one of the most used programming languages, there are huge amounts of legacy spreadsheets. In order to provide a MDE environment for such legacy spreadsheets, we have implemented in MDSHEET the model inference technique that we have proposed in [3].

The functionality of MDSHEET has furthermore been integrated in a widely used spreadsheet system. With our extension, we believe to provide a controlled environment for the safe development of spreadsheets, with opportunities for reducing errors in spreadsheets.

In the next three Sections, each of this features is briefly explained. The video that accompanies this paper is also divided in these same parts.

II. EMBEDDING CLASSSHEETS IN SPREADSHEETS

ClassSheets [4] are a high-level, object-oriented formalism to specify the business logic of spreadsheets. *ClassSheets* allow users to express business object structures within a spreadsheet using concepts from the UML. In fact in [7] we have described a technique to automatically map *ClassSheets* to UML.

Using the *ClassSheets* model, it is possible to define spreadsheet tables and to give them names, to define labels for the table's columns, to specify the types of the values such columns may contain and also the way the table expands (*e.g.*, horizontally or vertically).

Besides a textual (and formal) definition, *ClassSheets* also have a visual representation which very much resembles spreadsheets themselves [8]. We have embedded such visual model representation that mimics the well-known embedding of a domain specific language in a general purpose one. Like in such embeddings, we inherit all the powerful features of the host language: in our case, the powerful interactive interface offered by the (host) spreadsheet system. This approach has two key advantages: first, we do not have to build and maintain a complex interactive tool. Second, we provide *ClassSheet* model developers the programming environment they are used to: a spreadsheet environment. Furthermore,

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. PTDC/EIA-CCO/108613/2008. The three first authors were also supported by FCT grants SFRH/BPD/73358/2010, SFRH/BPD/46987/2008, BI4-2011_PTDC/EIA-CCO/108613/2008, respectively.

because the *ClassSheet* model and the spreadsheet data are defined in the same environment, we now have the power to ensure that both are synchronized.

In order to illustrate our embedding we present in Figure 1 an embedded *ClassSheet* model (Figure 1b) and one of its possible instances (Figure 1a).

	A	B	C
1	**Planes**		
2	**N-Number**	**Model**	**Name**
3	N2342	B 747	Magalhães
4	N341	B777	Cabral

(a) Planes table.

	A	B	C
1	**Planes**		
2	**N-Number**	**Model**	**Name**
3	n-number=""	model=""	name=""
4	⋮	⋮	⋮

(b) Planes *ClassSheet* model.

Figure 1. Planes example.

The *ClassSheet* model represents planes, where a plane is defined by an **N-Number**, **Model** and **Name**. In row 3, it is defined the type, and the default value, associated with each column: in this example all columns hold strings with an empty string as default value. The fourth row of the model contains vertical ellipses in all columns. This means that it is possible for these columns to expand vertically: the tables that conform to this model can have as many rows as needed.

Reusing the *ClassSheet* table we built before, we can now model a table to register concrete flights by the airline company, as shown in the right-hand side of Figure 2.

The colors in the model are used to distinguish the different entities represented, namely, *pilots*, *references to pilots* in the scheduling table, *reference to planes* in the scheduling table and the *flight scheduling* itself.[1]

A. Generating Spreadsheets from ClassSheet Models

Following the *Gencel* approach [9], the previously described models can be translated into initial spreadsheets together with tailor-made versions of update operations. These operations are defined to perform the tasks of insertion or deletion in such a way that the spreadsheet correctness is always preserved. In fact, the spreadsheet in the left-hand side of the Figure 2 was produced by our embedding mechanism: the initial spreadsheet will contain the labels in bold on the model, the initial formulas and buttons (grey rows/columns labelled with elipsis) to add new vertical or horizontal blocks of cells.

By using this MDE approach, the end user is guided in the introduction of data that conforms to the underlying model: for example, rows with type integer only accept integer values. Another key feature of this approach is that blocks of cells are automatically produced, for example, to add a new flight, which is a relationship between a pilot and a plane, the user must click on the button in row `6`. The system will then add a new row, also updating the necessary formulas: it will update the formulas in cells `E7`, `I7` and `K7` to include the new added row. This mechanism prevents the user from editing the spreadsheet without correctly updating its formulas, and therefore from corrupting it.

[1]For space limitations, we refrain from showing again the *planes* table.

III. Co-evolution of Models and Instances

One key advantage of using a model-driven software development process is the ability to interact both with the model (a *ClassSheet* in our case) and its instance (*i.e.*, the spreadsheet data). This is usually a complex task because the model and the instances need to be synchronized! In this section we present a set of spreadsheet co-evolution rules. Such rules define evolution steps for *ClassSheet* models and their instances and guarantee synchronization. These rules are specified using data-refinement theory which provides an algebraic framework for calculating with data types and corresponding values. It consists of type-level coupled with value-level transformations. The type-level transformations deal with the evolution of the model and the value-level transformations with the instances of the model (*e.g.* values).

We have designed an appropriate representation of spreadsheet models, including the fundamental notions of formulæand references [6]. For models and their instances, we have designed coupled transformation rules that cover specific evolution steps, such as the insertion of columns in all occurrences of a repeated block of cells. The rules are divided into three categories: *combinators*, used as helper rules, *semantic* rules, intended to change the model itself (*e.g.* add a column), and *layout* rules, designed to change the visual arrangement of data (*e.g.* swap two columns).

Combinator Rules: The first set of rules, *combinators*, include rules such as $after$, which means "apply the argument rule after the argument label". These combinators receive a rule as an argument and apply it in a specific place of the model, and thus, they are refinements or isomorphisms if the argument rule is a refinement or an isomorphism.

Semantic Rules: In [6] we have introduced a full catalog of spreadsheet evolution refinement rules. The catalog includes rules such as *make it expandable* that makes a block of cells expandable (horizontally or vertically), and *split* that moves a column to a new place and replaces it by references to the new locations. Their full definitions and Haskell implementations can be found in [6]. As an example, we graphically present in Figure 3 the *insert a column* rule:

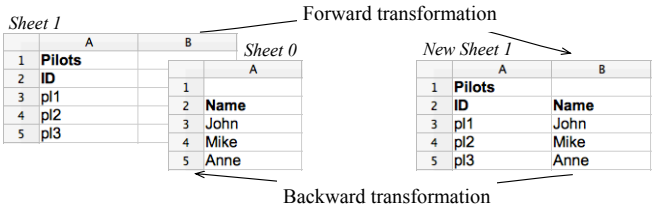

Figure 3. Adding/removing a column visually.

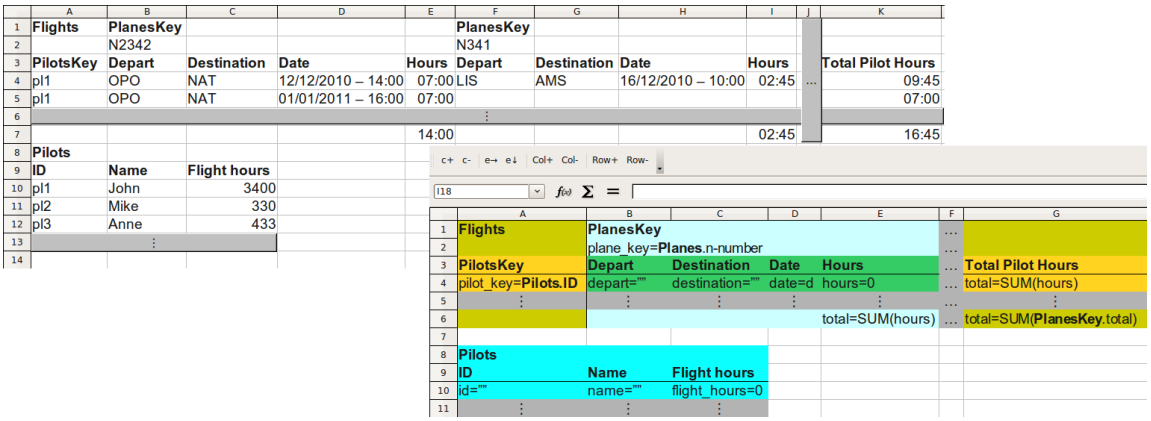

Figure 2. Spreadsheet of an airline company and an abstract model representing it.

Sheet 1 represents the original spreadsheet with an existing column which is transformed in *New Sheet 1* when applied the forward transformation. Applying the backward function, we get the original spreadsheet, *Sheet 1*, and a new sheet containing the removed column, *Sheet 0*. When applying the forward function *Sheet 0* is not used, and thus, not necessary to exist at that point in time.

The forward transformation, that is, to add a new column is available in the spreadsheet environment as the button `Col+` (*e.g.*, right-hand part of Figure 2) while the backward function, that is, to remove a column, in the `Col−` button.

Layout Rules: As the name suggests, *layout* rules are intended to change the arrangement of spreadsheets only, and not to add or remove any particular information. This set of rules includes evolution steps for changing the orientation of a spreadsheet from vertical to horizontal or to rearrange cells according to some conventions, for example.

Having defined a set of model evolution steps, we can now evolve the model and have the data automatically co-evolved. For example, in the *ClassSheet* of Figure 2, if we wish to add an attribute meal to a flight, we can do it in several steps: first, we add a column named **Meal** to the **Flights** class (in the last column of the dark green area, after column `E`). Second, we set its default value to ”NO”. Then, these model evolution steps are automatically reflected in the data, with a new column **Meal** being inserted by the framework in the two **Flights** instances in the data (new columns are added after column `E` and `I`). Also, the formulas of columns `K` are automatically updated.

IV. ClassSheet Model Inference

Consider the example spreadsheet shown in Figure 4. This spreadsheet represents again flights of a company, but in a less organized (yet probably more common) manner.

The business logic that underlies this spreadsheet is not immediately clear and is quite difficult to infer for a non-expert. In this section we will describe a strategy to infer such a business logic from the data in a spreadsheet.

	ID	Name	Flight Hours	N-Number	Model	Name	Depart	Destination	Date	Hours
2	pl1	John	3400	N2342	B 747	Magalhães	OPO	NAT	12/12/2010 – 14:00	7:00
3	pl1	John	3400	N2342	B 747	Magalhães	OPO	NAT	01/01/2011 – 16:00	7:00
4	pl3	Anne	433	N231	A 380	Camões	MAD	AMS	01/01/2010 – 18:00	2:30
5	pl1	John	3400	N341	B 777	Cabral	LIS	AMS	16/12/2010 – 10:00	2:45

Figure 4. A spreadsheet to store an airline company flights.

Objects that are contained in such a spreadsheet and the relationships between them are reflected by the presence of *functional dependencies* between spreadsheet columns.

It is possible to construct a relational model from a set of observed functional dependencies [3]. Such a model consists of a set of relations (each given by a set of column names) and expresses the basic business model present in the spreadsheet. Each relation of such a model results from grouping functional dependencies together. For example, for the spreadsheet in Figure 4 we can infer the following relational model (underlined columns indicate those on which the other columns are functionally dependent (primary keys); # indicates attributes referencing other table columns (foreign keys); tables between symbols <> represent relationships):

Pilots (ID, Name, Flight hours)
Planes (N-Number, Model, Name)
<Flights> (#ID, #N-Number, Depart, Destination, Date, Hours)

The model has two relations, Pilots and Planes, and a relationship connecting them, Flights. A relational model is very expressive, but it is not quite suitable for spreadsheets From this relational model we can generate the *ClassSheet* shown in Figure 2, as explained in [3].

The architecture of our approach is sketched in Figure 5.

V. Tool Architecture

In this section we present the architecture of our model-driven spreadsheet environment. In this environment, end users can interact both with the *ClassSheet* model and the spreadsheet data. Our techniques guarantee the synchronization of the two representations. In this setting, the

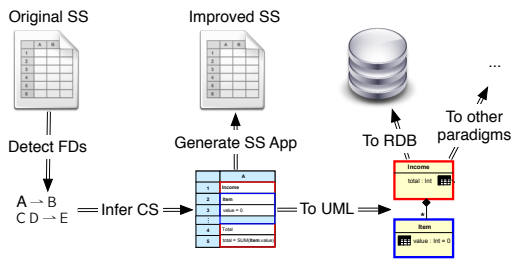

Figure 5. *ClassSheet* inference process.

spreadsheets consists of two sheets: `Sheet 0`, containing the embedded *ClassSheet* model and `Sheet 1`, containing the spreadsheet data that conforms to the model. We have defined an add-on to a widely used spreadsheet system, the *OpenOffice.org* system, so end users can evolve their models by using predefined buttons in the spreadsheet environment For each button, we defined a *OpenOffice.org* BASIC script that interprets the desired functionality, and send the contents of the spreadsheet (both the model and the data) to the MDSHEET framework. The MDSHEET framework was developed in HASKELL, and implements the co-evolution of the spreadsheet models and data. The global architecture of the tool we developed is presented in Figure 6.

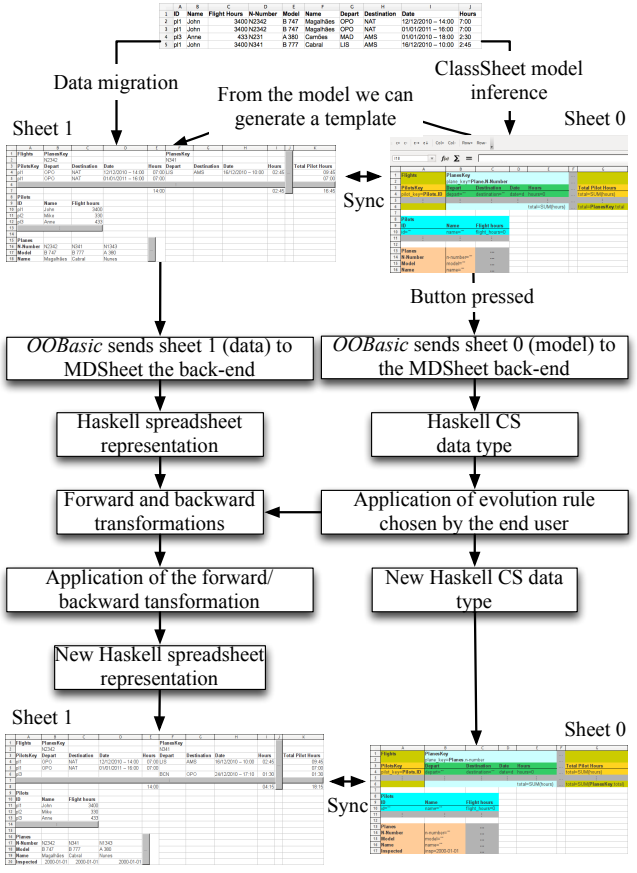

Figure 6. Spreadsheet model-driven environment.

In this environment users can build *ClassSheets* from scratch using the provided buttons. Also, we consider model inference from spreadsheet data [3]. This is particularly important when considering legacy spreadsheets. Moreover, the generated refactored spreadsheet includes some business logic rules (expressed as spreadsheet formulas) that assist end users in the safe and correct introduction/editing of data.

Tool and demonstration video availability: The MDSHEET tool and a video with a demonstration of its capabilities are available at the SSaaPP project web page: http://ssaapp.di.uminho.pt/twiki/bin/view/Main/Software.

VI. CONCLUSIONS

In this paper, we have proposed a framework for model-driven spreadsheet development. While having models and instances in the same environment potentiates further analysis, we expect most end users to interact with data instances only. In order to fully explore this situation, but also to realize the impact of our work in practice, we are already preparing an empirical study with human spreadsheet users.

REFERENCES

[1] R. Panko, "Facing the problem of spreadsheet errors," *Decision Line, 37(5)*, 2006.

[2] R. Abraham and M. Erwig, "Inferring templates from spreadsheets," in *Proc. of the 28th Int. Conf. on Software Engineering*. New York, NY, USA: ACM, 2006, pp. 182–191.

[3] J. Cunha, M. Erwig, and J. Saraiva, "Automatically inferring ClassSheet models from spreadsheets," in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, 2010, pp. 93–100.

[4] G. Engels and M. Erwig, "ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications," in *20th IEEE/ACM Int. Conf. on Automated Sof. Eng., Long Beach, USA*. ACM, 2005, pp. 124–133.

[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 Languages and Human-Centric Computing*. IEEE CS, 2011, pp. 186–201.

[6] J. Cunha, J. Visser, T. Alves, and J. Saraiva, "Type-safe evolution of spreadsheets," in *Int. Conf. on Fundamental Approaches to Software Engineering*, ser. FASE'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 186–201.

[7] J. Cunha, J. P. Fernandes, and J. Saraiva, "From Relational ClassSheets to UML+OCL," in *SAC'12: the Software Engineering Track at the 27th Annual ACM Symposium On Applied Computing*. ACM, March 2012, pp. 1151–1158.

[8] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger, "Automatic generation and maintenance of correct spreadsheets," in *Proc. of the 27th Int. Conf. on Software Eng.* New York, NY, USA: ACM, 2005, pp. 136–145.

[9] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert, "Visual specifications of correct spreadsheets," in *IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, 2005, pp. 189–196.