Embedding and Evolution of Spreadsheet Models in Spreadsheet Systems

Jácome Cunha, Jorge Mendes and João Saraiva
Universidade do Minho, Portugal
jacome@di.uminho.pt jorgecunhamendes@gmail.com
jas@di.uminho.pt

João Paulo Fernandes
Universidade do Minho & Universidade do Porto, Portugal
jpaulo@{di.uminho.pt,fe.up.pt}

***Abstract*—This paper describes the embedding of *ClassSheet* models in spreadsheet systems. *ClassSheet* models are well-known and describe the business logic of spreadsheet data. We embed this domain specific model representation on the (general purpose) spreadsheet system it models. By defining such an embedding, we provide end users a model-driven engineering spreadsheet developing environment. End users can interact with both the model and the spreadsheet data in the same environment. Moreover, we use advanced techniques to evolve spreadsheets and models and to have them synchronized. In this paper we present our work on extending a widely used spreadsheet system with such a model-driven spreadsheet engineering environment.**

I. Introduction

In recent years the spreadsheet research community has recognized the need to support *end-user model-driven software development*, and to provide spreadsheet developers and end users with methodologies, techniques and the necessary tool support to improve their productivity. Along these lines, several techniques have been proposed, being spreadsheet templates [1], *ClassSheets* [2] and the use of class diagrams to specify spreadsheets [3] fundamental contributions. These proposals guarantee that end users safely edit their spreadsheets and they introduce a form of model-driven software development: a spreadsheet business model is defined from which a customized spreadsheet application is generated guarantying the consistency of the spreadsheet with the underlying model.

Despite of its huge benefits, model-driven software development is sometimes difficult to realize in practice. In the context of spreadsheets, for example, the use of model-driven software development requires that the developer is familiar both with the spreadsheet domain (business logic) and with model-driven software development. In the particular case of the use of templates, a new tool is necessary to be learned, namely *Gencel* [4]. Using this tool, it is possible to generate a new spreadsheet respecting the corresponding model. This approach, however, has several drawbacks: first, in order to define a model, spreadsheet model developers will have to become familiar with a new programming environment. Second, and most important, there is no connection between the stand alone model development environment and the spreadsheet system. As a result, it is not possible to (automatically) synchronize the model and the spreadsheet data. That it is to say that the co-evolution of the model and its instance is not possible.

In this paper, we propose to embed of *ClassSheet* spreadsheet models in spreadsheets themselves. Our approach closes the gap between creating and using a domain specific language for spreadsheet models and a totally different framework for actually editing spreadsheet data. Instead, we unify these operations within spreadsheets: in one sheet we define the underlying model while another sheet holds have the actual data, such that the model and the data are kept synchronized by our framework. Model evolution is available as a set of predefined operations, and changes in a model are automatically propagated to its data. Our framework is such that editing spreadsheet data is also safe and controlled.

The paper is organized as follows: in section II we present an embedding of the *ClassSheet* language in a generic spreadsheet environment. In section III we present formal rules to support the evolution of spreadsheets. Section IV describes the architecture of the method we propose for spreadsheet evolution which we demonstrate in section V using practical examples. In section VI we briefly discuss the evaluation of a model-driven spreadsheet development approach. In section VII we presented related work and finally, in section VIII we draw our conclusions.

II. The *ClassSheet* Model and its Embedding

ClassSheets [2] 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 Unified Modeling Language (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 [4]. Thus, such visual model representation makes developing the spreadsheet model very similar to creating a concrete spreadsheet. In order to support this visual representation, a specific interactive tool has been developed [1]. This tool provides a powerful interactive environment to create *ClassSheets* and to automatically generate

spreadsheets that follow the specified business model. The generated spreadsheets guide end users in introducing data that follows the underlying model, thus avoiding several common (spreadsheet) errors.

In this section we present an embedding of *ClassSheet* models in spreadsheet systems. In this embedding we mimic 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, because the *ClassSheet* model and the spreadsheet data are defined in the same environment, we now have the power to ensure that the they are synchronized.

Before we present the architecture of our model-driven spreadsheet programming environment, and the techniques we use to synchronize the *ClassSheet* model and the spreadsheet data, let us present our embedding of *ClassSheet* models.

A. Vertically Expandable Tables

In order to illustrate our approach we shall consider an example modeling an airline scheduling system which we adapted from [5]. We assume that any airline company must record the activity of its pilots in, typically, a software system. A simple way of achieving this goal is to use a spreadsheet, and a table as the one presented in Figure 1a. This table has a title, **Pilots**, and a row with labels, one for each of the table's column: **ID** represents a unique pilot identifier, **Name** represents the pilot name and **Flight hours** represents the total number of hours a pilot has already flown. Each of the subsequent rows represents a concrete pilot.

	A	B	C
1	**Pilots**		
2	**ID**	**Name**	**Flight hours**
3	pl1	John	3400
4	pl2	Mike	330
5	pl3	Anne	433

(a) Pilot table.

	A	B	C
1	**Pilots**		
2	**ID**	**Name**	**Flight hours**
3	id=""	name=""	flight_hours=0
4	⋮	⋮	⋮

(b) Pilot *ClassSheet* model.

Fig. 1: Pilot example.

Tables such as the one presented in Figure 1a are frequently used within spreadsheets, and it is fairly simple to create a model specifying such tables. For the example shown, we can extract the model presented in Figure 1b.

To model the labels we use a textual representation and the exact same names as in the data sheet (**Pilots**, **ID**, **Name** and **Flight hours**). To model the actual data we abstract concrete column cell values by using a single identifier: we use the one-worded, lower-case equivalent of the corresponding column label (so, *id*, *name* and *flight_hours*). Next, a type is associated with each column: columns A and B hold strings (denoted in the model by the empty string "" following the = sign), and column C holds integer values (denoted by 0 following =). Notice that the fourth row of the model contains vertical ellipses in all columns. This means that it is possible for this column to expand vertically: the tables that conform to this model can have has many rows as needed. The scope of the expansion is between the ellipsis and the black line (between row 2 and 3). Note that, by definition, *ClassSheets* do not allow for nested expansion blocks, and thus, there is no possible ambiguity associated with this feature.

The instance shown in Figure 1b, for example, has three pilots. All the model cells shown are colored with the same color meaning that they are part of the same table[1].

B. Horizontally Expandable Tables

In the lines of what we described in the previous section, airline companies must also store information on their airplanes. This would be the purpose of table **Planes** in the spreadsheet illustrated in Figure 2a, which is organized as follows: the first column holds labels that identify each row, namely, **N-Number**, **Model** and **Name**; cells in row **N-Number** (respectively **Model** and **Name**) contain the unique n-number identifier of a plane, (respectively the model of the plane and the name of the plane). Each of the subsequent columns contains information about one particular aircraft.

	A	B	C	D
1	**Planes**			
2	**N-Number**	N2342	N341	N1343
3	**Model**	B 747	B 777	A 380
4	**Name**	Magalhães	Cabral	Nunes

(a) Plane table.

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

(b) Plane *ClassSheet* model.

Fig. 2: Plane example.

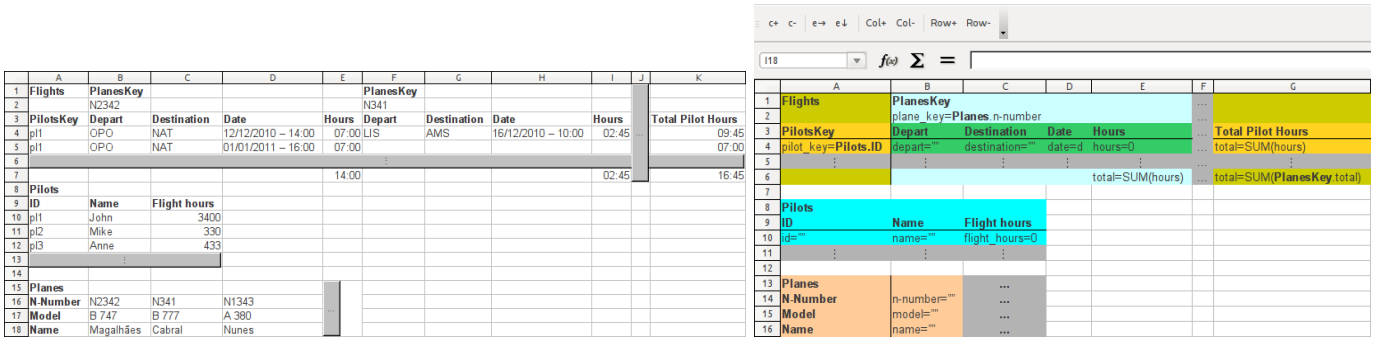
The **Planes** table can be modelled by the illustration in Figure 2b. This model may be constructed following the same strategy as in the previous section, but now swapping columns and rows: the first column contains the label information and the second column the names abstracting concrete data values: again, each cell has a name and the type of the elements in that row (in this example, all the cells are to hold strings); the third column has ellipses meaning that rows are horizontally expandable. Notice that the instance table has information about three planes.

C. Relationship Tables

In this section, we explore an example of more practical interest than the two shown before (which are admittedly simple). So far, we have modelled tables for pilots and planes. Reusing what we built, we can now model, as shown in Figure 3a, a table for registering concrete flights by the airline company.

We can see the same information we had before: between rows 8 and 12, we have the pilot information shown in Figure 1a and between rows 14 and 17 the plane information shown in Figure 2a. The lines between rows 1 and 6 represent

[1]We assume colors are visible in the digital version of this paper.



(a) Spreadsheet instance of an airline company.

(b) Embedded *ClassSheet* representing a system for an airline company.

Fig. 3: Spreadsheet of an airline company and an abstract model representing it.

the desired scheduling information. For simplicity, let us for now focus on columns `A` to `E`. Column `A` holds the identifier of the pilot for a concrete flight. Row 2, columns `B` and `F`, hold the identifiers of the airplanes assigned to fly from *OPO* to *NAT*, two times, and from *LIS* to *AMS*, respectively. Origins and destinations of flights are registered in **Depart** and **Destination** columns, as well as the date and hour of departure **Date** and the number of hours the flight will take **Hours**. Notice that we can have as many entries for pilots (planes, respectively) as we need just by adding one row per pilot (and 4 columns per plane). An example of the way we read this table is as follows:

pilot pl1 flew plane N2342 from OPO to NAT on December 12th, 2010, 14:00 hours and the flight took 7 hours.

The *ClassSheet* model illustrated in Figure 3b abstracts, in a very straightforward way, the data instance that we have just described. The two bottom blocks of cells represent the same *ClassSheet* that we have shown before for pilots and planes. The top block may expand both vertically and horizontally as indicated by the ellipses. The vertical expansion is necessary to add more pilots; the horizontal is used to add more planes.

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

The *ClassSheet* presented in Figure 3b looks very much like the visual representation introduced in [2] (see, for example, Figure 7 of that paper). Like in any other embedding of a DSL in a host language, however, we also suffer here from some syntactic limitations: for example, when horizontally expanding a *ClassSheet* model the corresponding column identifier separators are not shown (for example, the vertical bar between columns `B` and `C`). Due to the before mentioned host language restrictions, we have no way of doing this in our embedding.

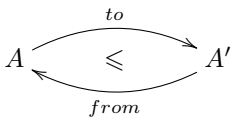
D. Generating Spreadsheets from ClassSheet Models

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. The model presented in Figure 3b can be used to generate the spreadsheet in Figure 3a (this spreadsheet has already been edited after the initial generation). The initial spreadsheet will contain the labels in bold on the model, the initial formulas and buttons to add new vertical or horizontal blocks of cells. For example, in the **Pilots** table, there is a button on row `13` which will insert a new row. The values that will appear in the new row are the default values defined in the model and the user can only update them to a value of the same type (string, integer, etc.). A more complex example is to add a new flight which is a relationship between a pilot and a plane, plus some more information. If the user clicks on the button in row `6`, the system will add a new row as explained before, but in this case it will also update the necessary formulas: it will update the formulas in cells `E7`, `I7` and `K7` to include the new added row. This mechanism, also used in the *Gencel* [4] framework, prevents the user from editing the spreadsheet without correctly updating its formulas, and therefore from corrupting it. The button in column `J` works in a similar way, updating the formulas in cells `K4` and `K5`.

III. Evolution Rules

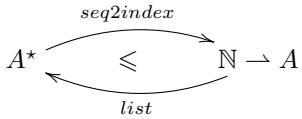
One key advantage of using a model-driven software development process is the ability to interact both with the model (a *ClassSheet* in our case) and its instance (*i.e.*, the spreadsheet data). This is usually a complex task because the model and the instances need to be synchronized! In this section we present a set of co-evolution rules common is spreadsheets. Such rules define evolution steps for *ClassSheet* models and its instances and they guarantee synchronization. These rules are specified using data refinements theory which provides an algebraic framework for calculating with data types and corresponding values [6]–[8]. 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 deal with the instances of the model (*e.g.* values). The next diagram, where A and A' are the original and the transformed representations of a data type, respectively, depicts the general

scenario of a transformation in this framework:



Each transformation is coupled with a witness injective function to, also called *forward transformation*, of type $A \to A'$ and a witness surjective function $from$, also called *forward transformation*, of type $A' \to A$. These functions are responsible for converting values of type A into type A' and back. If a refinement works in both directions, that is, if $A \leqslant A'$ and $A' \leqslant A$, then we have an isomorphism, and we say that $A \cong A'$.

A common example of a refinement [9] shows that finite maps are the implementation for lists: finite maps from natural numbers to some type A, $\mathbb{N} \rightharpoonup A$, are the implementation of lists of that type, $A^\star$:



Function $seq2index$ creates a finite map where the keys are the indexes of the elements of the list. Function $list$ collects the elements in the map. For example,

$$seq2index\ [\texttt{'a'},\texttt{'z'}] = \{1 \mapsto \texttt{'a'}, 2 \mapsto \texttt{'z'}\}$$
$$list\ \{1 \mapsto \texttt{'a'}, 2 \mapsto \texttt{'z'}\} = [\texttt{'a'},\texttt{'z'}]$$

The *Two Level Transformation* (2LT)[2] framework is a HASKELL implementation of this data refinement theory [9]–[11]. It provides the basic combinators to define and compose transformations for data types and witness functions. Since 2LT is statically typed, transformations are guaranteed to be type-safe ensuring consistency of data types and data instances. Frameworks like 2LT are considered bidirectional transformation systems [12], [13].

In previous works we have designed an appropriate representation of spreadsheet models, including the fundamental notions of formula and references [14], [15]. For these models and their instances, we have designed coupled transformation rules that cover specific spreadsheet evolution steps, such as the insertion of columns in all occurrences of a repeated block of cells. Each model-level transformation rule is coupled with instance level migration rules from the source to the target model and vice versa. These coupled rules can be composed to create compound transformations at the model level inducing compound transformations at the instance level. This approach guarantees safe evolution of spreadsheets even when models change.

The rules presented in [14], [15] guarantee that forward transformations fully preserve data information. This is, however, not the case for backward transformations. For example, the transformation to insert a column in a spreadsheet is a forward step with the corresponding backward step being the removal of that column. This backward step being applied will permanently delete the respective column, with all its information being lost. Information is lost due to the fact that data refinements need not be isomorphisms. In this paper, we consider data refinements that are, in fact, isomorphisms: we guarantee that data is never lost, either in forward or backward transformations. This ability will allow us to fully do and undo any transformation, in either direction.

The rules are divided in three categories: *combinators*, used as helper rules, *semantic* rules, intended to change the model itself (*e.g.* add a new column), and *layout* rules, designed to change the visual arrangement of the spreadsheet (*e.g.* swap two columns).

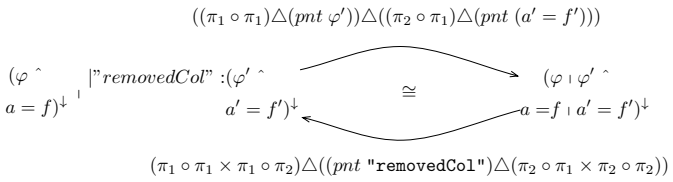
A. Combinator Rules

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.

B. Semantic Rules

In previous versions of our evolution rules [14], [15], the ones classified as *semantic* are refinements *only*, since their backward transformation loses information. To avoid this, we now redefine them as new rules that do not lose data, that is, as isomorphisms.

a) Insert a column: A column is defined as three vertically aligned cells: the first with the label, the next with the definition of its rows and the last with the ellipsis indicating that more rows can be added. Figure 4 represents such a model in column A. Formally, a column is defined as $(\varphi \hat{}\ a = f)^{\downarrow}$, where φ is the label, $\hat{}$ means that it is a vertical composition and $a = f$ is the definition of the rows, that is, a field named a and with definition f, which can be a plain value or a spreadsheet formula. The diagram shown next formally represents this rule:



Note that the column definition, $(\varphi \hat{}\ a = f)^{\downarrow}$, is broken in two lines so it looks more like it will be in the spreadsheet. The right-hand side of the rule is defined by an existing column on which right-hand side will be placed the newly added column, defined as $(\varphi' \hat{}\ a' = f')^{\downarrow}$. The left-hand side of the rule is defined by the existing column on which right-hand side will be placed a new sheet containing the removed column. For an explanation of the to and $from$ functions, the reader is referred to [14].

[2]Available at http://code.google.com/p/2lt.

To apply the rule from the left to the right (add the column) it seems necessary to have already the column we want to add, but, in fact, this is not true: only its type is needed. When the function to migrate the data is applied, it never uses that part of the data, and thus, it does not need to exist. This rule is visually represented in Figure 4:

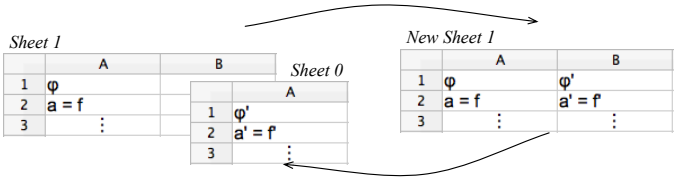


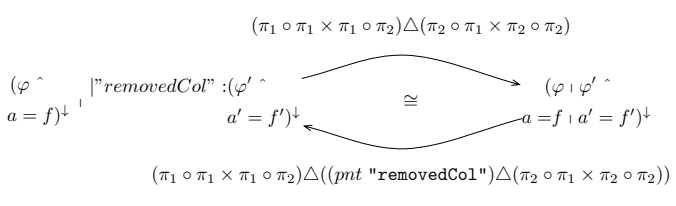
Fig. 4: Adding/removing a column visually.

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.*, Figure 3b) while the backward function, that is, to remove a column, in the `Col−` button. If the button to add a column is pressed, the environment will ask the user for a label and for a field name and a default value, as well as an existing column label after which it should place the new column. This information is sent to the HASKELL back-end which will return the new model so the front-end changes the exiting one.

This rule prevents the removed column from being lost, which, in principle, would allow to recover it, but this is not possible using this rule. Remember that it uses external input to create the new column. We will now present a new rule that allows this recovery.

b) Recover a deleted column: The rule to recover a deleted column is similar to the previous one, but it does not need external information, that is, the label and row definition. The next diagram formalizes such rule:



In fact, the only change from one to the other is the *to* function: to recover an existing column, no external input is need; instead, the extra (deleted) column is used. The backward transformation removes the column and stores it again is an auxiliary place.

Although these rules are very similar, to the end user using the spreadsheet advanced system, they are quite different. When using the first, the user intends to add or remove a column; when using the second, the user wants to recover a previous existing and deleted row. In fact, the user can only use the second rule after using the first; obversely one can only recover something after deleting it.

c) Other rules: In [14], [15] 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. There full definitions and HASKELL implementations can be found in [15]. Because such definitions are not needed to understand our techniques, we omit them here.

C. 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. These rules have already been defined as isomorphisms in our previous work, and thus, they can be used directly in our embedding.

IV. MODEL-DRIVEN SPREADSHEET PROGRAMMING

Having defined an embedding of the *ClassSheet* model in a spreadsheet system, we present now the global 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 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 that end users can evolve their models by using predefined buttons in the spreadsheet environment (see Figure 4). 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 HaExcel framework. The HaExcel 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 5.

In this environment end users can build the *ClassSheet* from scratch using the provided buttons. However, we consider also the inference of the model from the spreadsheet data [16]. This is particularly important when we are considering legacy spreadsheets. Moreover, the generated refactored spreadsheet includes some business logic rules (expressed as spreadsheet formula) that assist end users in the safe and correct introduction/edition of data. For example, in the column with label

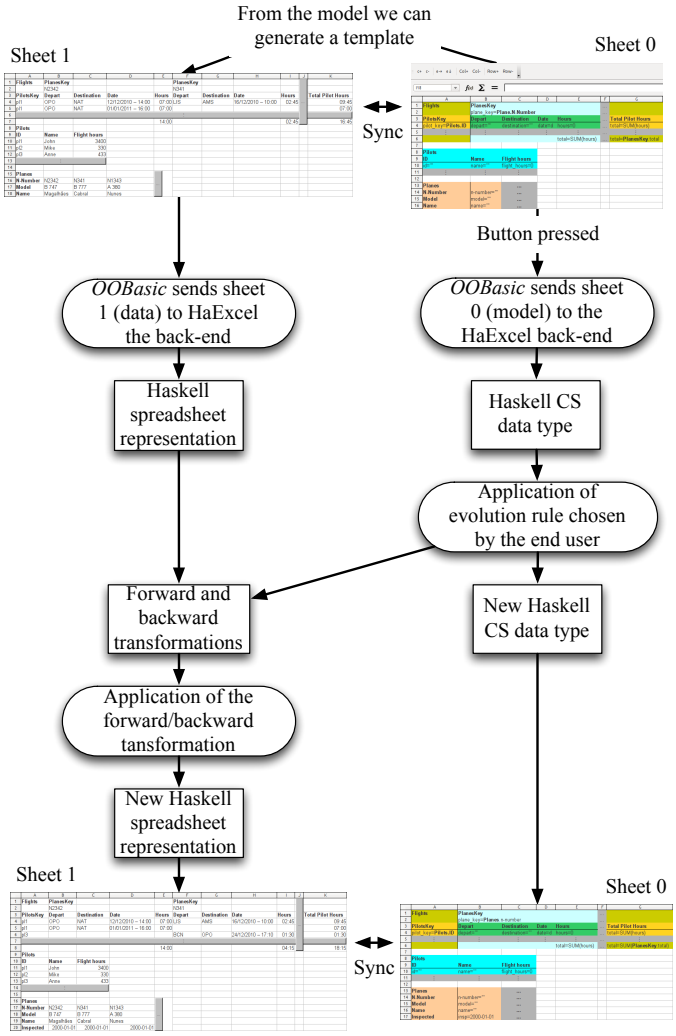


Fig. 5: Architecture of the spreadsheet model-driven environment.

Flight Hours in Figure 3, the user can only introduce integer values.

The interdependencies between spreadsheet data can be captured by a powerful mechanism called *functional dependencies*, a concept from relational databases theory describing the relationship between attributes of a table [17]. From the set of functional dependencies inferred before, we can devise a *ClassSheet* model, as we explain in detail in [16].

V. Evolution

In this section we illustrate with concrete examples the evolution operations that are available in our tool. Starting with a pair (spreadsheet, model) like the one in Figure 3, the user can perform several actions to modify both the spreadsheet and the model.

For example, clicking the button that occupies spreadsheet's line 6 in its entirety causes an evolution on the spreadsheet itself: a new row is inserted. The new row is available where the button used to be, the button is shifted one line down and the formulas defined in cells `E7` and `I7` (Figure 3a) are updated: their new value is calculated also with the values in cells `E6` and `I6`, which the use may insert (together with any other cell value in line 6). We can see the result of this operation in Figure 6, with a new row for pilot *pl3* (who is flying plane *N341*); the model remains the original one since the change operated leads to no structural change on the spreadsheet.

It is also possible to evolve the model, so that changes to the structure of the spreadsheet instance can be performed. A simple change that can be made is the addition of the date a plane was mechanically inspected by the last time. To do that, the user has to select the **Planes** class in the model and add a new row by clicking the *Row+* button given by our tool. Given the new row the name **Inspected**, we obtain the model presented in Figure 7. We can see a new expandable row (`17`), with a label and a data cell *inspected* with default value *2000-01-01*. The spreadsheet instance is then synchronized automatically in order to comply with the model: a new row is added to the planes table and the data cells have their value set to the default one as defined in the model. This is also show in that same Figure.

The evolution operations describe so far are quite simple but more complex ones can be performed. For example, in Figure 8, we can see the result of inserting a new column **N-Passengers** in the table **Flights** from the model in Figure 7. The new column has an expandable cell *passengers* with default value *0*. This modification is straightforward at the model level, but its propagation to the spreadsheet requires the addition of the a column to each group of column expansions, and the formulas to be updated to reference the correct cells. The spreadsheet instance in Figure 8 differs from the one in Figure 7 in that a column **N-Passengers**, with values set to 0, was added after every **Hours** column.

VI. Evaluation

In [15] and [18] we have shown that the use of models and a model-driven approach to spreadsheet development can help, in some cases, end users to be more effective and efficient. We organized an empirical study with real end users and ask them to performs several tasks that should benefit from the use of models. Moreover, we believe that professional programmers using spreadsheets should benefit even more from this approach. In fact, we are preparing a new study to evaluate the impact of spreadsheet models when used by professionals.

VII. Related Work

Ko *et al.* [19] summarize and classify the research challenges of the end-user software engineering area. This include requirements gathering, design, specification, reuse, testing and debugging. However, besides the importance of Lehman's laws of software evolution [20], very little is stated with respect to spreadsheet evolution. Spreadsheet evolution poses challenges not only in the evolution of the underlying model, but also in the migration of the spreadsheet values and the used

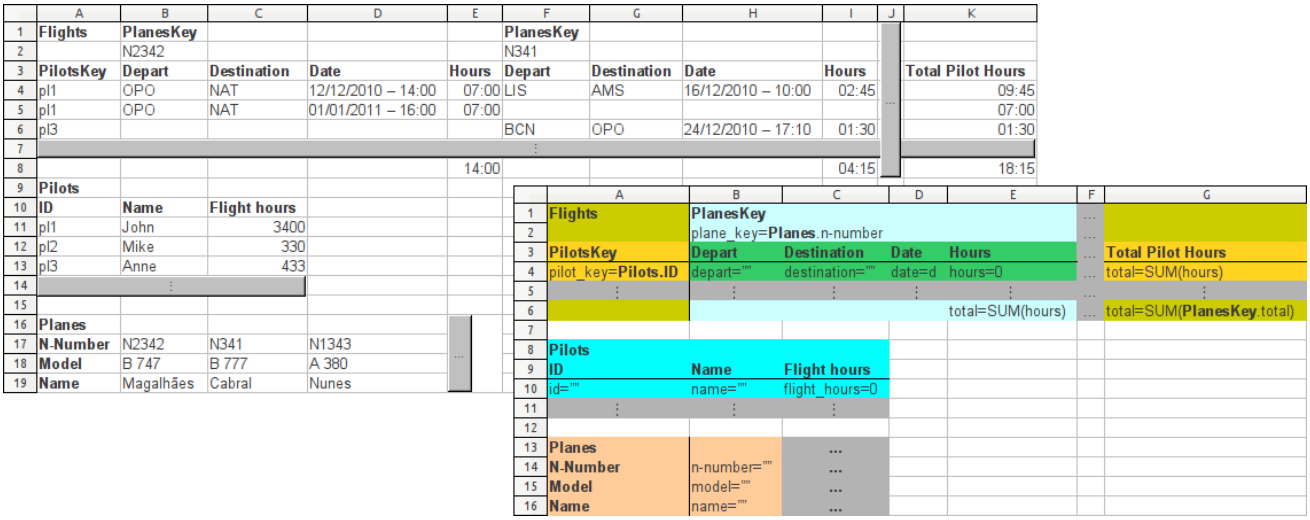


Fig. 6: Evolution – addition of a row to the flights in the spreadsheet.

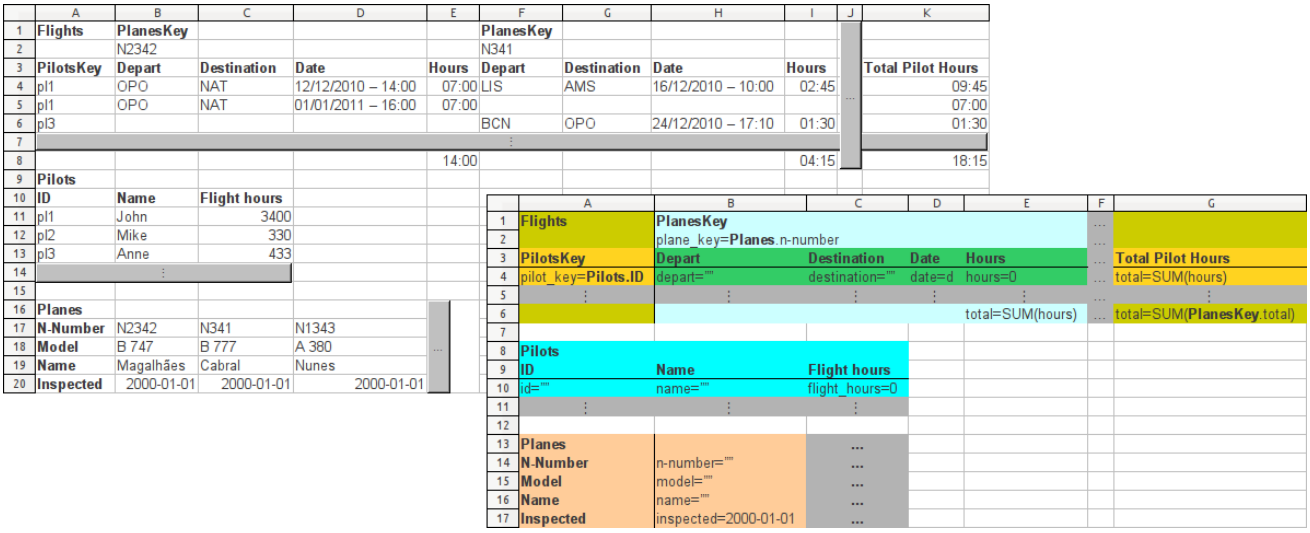


Fig. 7: Evolution – addition of a row in the planes table of the model.

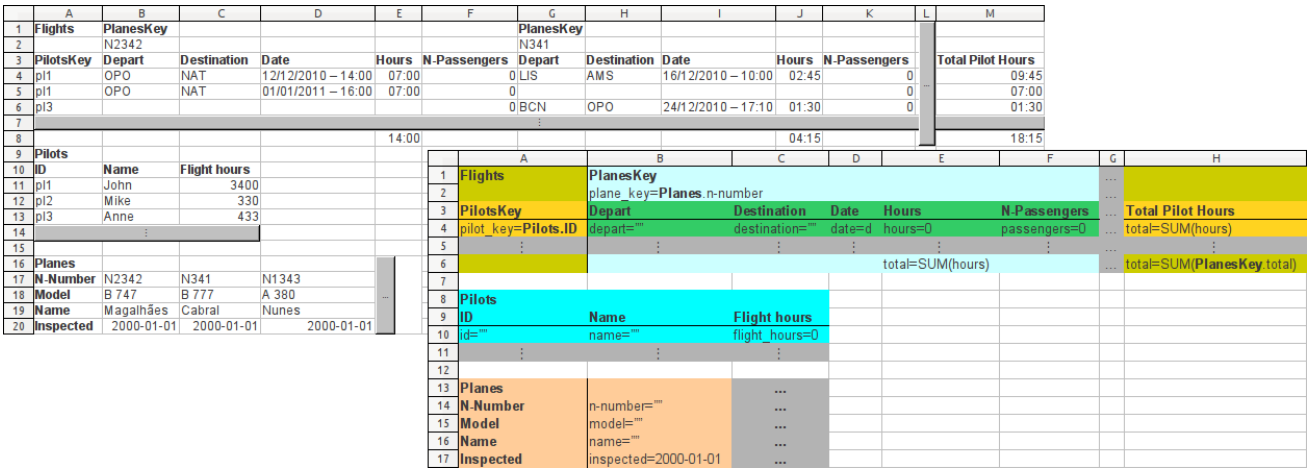


Fig. 8: Evolution – addition of a column in the flights table of the model.

formulæ. Nevertheless, many of the transformations applied within spreadsheet originate in works aiming at spreadsheet generation.

Engels *et al.* [21] propose a first attempt to solve the problem of spreadsheet evolution. ClassSheets are used to specify the spreadsheet model and transformation rules are defined to enable model evolution. These model transformations are propagated to the model instances (spreadsheets) through a second set of rules which update the spreadsheet values. The authors present a set of rules and a prototype tool to support these changes. In this paper we present a more advanced way to evolve spreadsheet models and instances in a different way: first, we use strategic programming with two-level coupled transformation. This enables type-safe transformations, offering guarantee that in any step semantics is preserved. Also, the use of 2LT not only gives us the data migration for free but it also allows back portability, that is, it allows the migration of data from the new model back to the old one. Moreover, we reuse the spreadsheet environment so the user does not need to learn a new tool/environment.

Vermolen and Visser [22] proposed a different approach for coupled evolution of data model and data. From a data model definition, they generate a domain specific language (DSL) which supports the basic transformations and allows data model and data evolution. The interpreter for the DSL is automatically generated making this approach operational. In principle, this method could also be used for spreadsheet evolution. However, while their approach is tailored for forward evolution, our own supports reverse engineering, that is, it supports automatic transformation and migration from a newer model to an older one.

VIII. Conclusions

In this paper, we have presented techniques and a tool for providing a model-driven engineering software development for spreadsheet programming. We have presented the embedding of a domain specific model representation in a widely used spreadsheet system. We have also presented techniques to perform co-evolution of the *ClassSheet* model and spreadsheet data. We have developed an extension for a widely used spreadsheet system where such embeddings and co-evolution rules are available. As future work, we plan to assess the impact of this approach on end user productivity by performing an usability study.

References

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

[2] G. Engels and M. Erwig, "ClassSheets: Automatic generation of spreadsheet applications from object-oriented specifications," in *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2005, pp. 124–133.

[3] F. Hermans, M. Pinzger, and A. van Deursen, "Automatically extracting class diagrams from spreadsheets," in *ECOOP '10: Proceedings of the 24th European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 52–75.

[4] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger, "Automatic generation and maintenance of correct spreadsheets," in *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM, 2005, pp. 136–145.

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

[6] C. Morgan and P. H. B. Gardiner, "Data refinement by calculation," *Acta Informatica*, vol. 27, pp. 481–503, January 1990.

[7] J. N. Oliveira, "A reification calculus for model-oriented software specification," *Formal Aspects of Computing*, vol. 2, no. 1, pp. 1–23, 1990.

[8] J. N. Oliveira, "Transforming data by calculation," in *GTTSE 2007*, ser. Lecture Notes in Computer Science, R. Lämmel, J. Visser, and J. Saraiva, Eds., vol. 5235. Springer, 2008, pp. 134–195.

[9] A. Cunha, J. N. Oliveira, and J. Visser, "Type-safe two-level data transformation," in *Proceedings of the 14th International Symposium on Formal Methods Europe*, ser. LNCS, J. Misra *et al.*, Eds., vol. 4085. Springer, 2006, pp. 284–299.

[10] T. L. Alves, P. F. Silva, and J. Visser, "Constraint-aware Schema Transformation," in *The Ninth International Workshop on Rule-Based Programming*, 2008.

[11] A. Cunha and J. Visser, "Strongly typed rewriting for coupled software transformation," *Electronic Notes on Theoretical Computer Science*, vol. 174, pp. 17–34, April 2007.

[12] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *ICMT '09: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 260–283.

[13] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt, "Boomerang: resourceful lenses for string data," in *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2008, pp. 407–419.

[14] J. Cunha, J. Visser, T. Alves, and J. Saraiva, "Type-safe evolution of spreadsheets," in *FASE'11: Procs. of the 13th Int. Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011*. Springer-Verlag, 2011, to appear.

[15] J. Cunha, "Model-based spreadsheet engineering," Ph.D. dissertation, University of Minho, March 2011.

[16] J. Cunha, M. Erwig, and J. Saraiva, "Automatically inferring ClassSheet models from spreadsheets," in *VLHCC '10: Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 93–100.

[17] J. Cunha, J. Saraiva, and J. Visser, "From spreadsheets to relational databases and back," in *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. New York, NY, USA: ACM, 2009, pp. 179–188.

[18] L. Beckwith, J. Cunha, J. P. Fernandes, and J. Saraiva, "End-users productivity in model-based spreadsheets: An empirical study," in *IS-EUD '11: Proceedings of the Third International Symposium on End-User Development*, 2011, to appear.

[19] A. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrence, H. Lieberman, B. Myers, M. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *Journal ACM Computing Surveys*, 2009.

[20] M. M. Lehman, "Laws of software evolution revisited," in *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*. London, UK: Springer-Verlag, 1996, pp. 108–124.

[21] M. Luckey, M. Erwig, and G. Engels, "Systematic evolution of typed (model-based) spreadsheet applications," submitted for publication.

[22] S. D. Vermolen and E. Visser, "Heterogeneous coupled evolution of software languages," in *MODELS'08: Procs. of the 11th Int. Conf. on Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, Eds., vol. 5301. Springer, September 2008, pp. 630–644.