Bidirectional Transformation of Model-Driven Spreadsheets*

Jácome Cunha[1], João P. Fernandes[1,2], Jorge Mendes[1], Hugo Pacheco[1], and João Saraiva[1]

[1] HASLab / INESC TEC, Universidade do Minho, Portugal
[2] Universidade do Porto, Portugal
{jacome,jpaulo,jorgemendes,hpacheco,jas}@di.uminho.pt

Abstract. Spreadsheets play an important role in software organizations. Indeed, in large software organizations, spreadsheets are not only used to define sheets containing data and formulas, but also to collect information from different systems, to adapt data coming from one system to the format required by another, to perform operations to enrich or simplify data, etc. In fact, over time many spreadsheets turn out to be used for storing and processing increasing amounts of data and supporting increasing numbers of users. Unfortunately, spreadsheet systems provide poor support for modularity, abstraction, and transformation, thus, making the maintenance, update and evolution of spreadsheets a very complex and error-prone task.
We present techniques for model-driven spreadsheet engineering where we employ bidirectional transformations to maintain spreadsheet models and instances synchronized. In our setting, the business logic of spreadsheets is defined by *ClassSheet* models to which the spreadsheet data conforms, and spreadsheet users may evolve both the model and the data instances. Our techniques are implemented as part of the MDSheet framework: an extension for a traditional spreadsheet system.

Keywords: Software Evolution, Data Evolution, Bidirectional Transformation, Model Synchronization, Spreadsheets

1 Introduction

Spreadsheets are widely used in the development of business applications. Spreadsheet systems offer end users a high level of flexibility, making initiation easier for new users. This freedom, however, comes at a price: spreadsheets are notoriously error prone as shown by numerous studies reporting that up to 90% of real-world spreadsheets contain errors [21].

* This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, projects ref. PTDC/EIA-CCO/108613/2008 and PTDC/EIA-CCO/120838/2010. The three first authors were also supported by FCT grants SFRH/BPD/73358/2010, SFRH/BPD/46987/2008 and BI4-2011PTDC/EIA-CCO/108613/2008, respectively.

In recent years, the spreadsheet research community has recognized the need to support end-user *model-driven spreadsheet development* (MDSD), 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, namely the use of templates [1], *ClassSheet* models [10] and class diagrams [14]. These proposals guarantee that end users can safely edit their spreadsheets and introduce a form of model-driven software development: they allow to define a spreadsheet business model from which a customized spreadsheet application holding the actual data is generated. The consistency of the spreadsheet data with the overlying model is guaranteed, often by limiting the editing options on the data side to ensure that the structure of the spreadsheet remains unchanged.

A significant drawback of such approaches lies in the fact that the evolution of both spreadsheet models and the instances generated from them is considered in isolation. That is to say that, after obtaining a spreadsheet instance from a particular model, a simple evolution step on the model side may break the conformity with its instance, and vice versa. A first attempt to overcome this limitation was proposed in [7], where it was shown how to co-evolve spreadsheet instances upon a model evolution defined according to a well-behaved set of possible transformations. The approach presented in [7], however, has two important drawbacks: *i)* the evolutions that are permitted at the model level can only be refinement steps: it is not possible to perform model evolutions that are frequent in spreadsheets such as removing a column, for example; and *ii)* it does not allow users to directly evolve spreadsheet instances having the corresponding model automatically co-evolved.

The goal of this paper is to study a more general setting where editing operations on models can be translated into conforming editing operations on spreadsheets and editing operations on spreadsheets can be translated into respective editing operations on models. For this purpose, we develop independent editing languages for both models and spreadsheets and bind them together using a symmetric bidirectional framework [9, 16] that handles the edit propagation. Among other properties, the fundamental laws governing the behavior of such bidirectional transformations guarantee that the conformity of spreadsheet instances and models can always be restored after a modification. Both the model and instance evolution steps are available as an extension of OpenOffice.

2 *ClassSheets* as Spreadsheet Models

Erwig *et al.* [10] introduced the language of *ClassSheets* to model spreadsheets at a higher abstraction level, thus allowing for spreadsheet reasoning to be performed at the conceptual level. *ClassSheets* have a visual representation very similar to spreadsheets themselves: in Figure 1, we present a possible model for a **Budget** spreadsheet, which we adapted from [10].[3]

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

	A	B	C	D	E	F
1	**Budget**	**Year**	year=2010		...	
2	**Category**	Qnty	Cost	Total	...	Total
3	name=""	qnty=0	cost=0	total=qnty*cost	...	total=SUM(total)
4	...	...	...	...	...	...
5				total=SUM(total)	...	total=SUM(year.total)

Fig. 1: Budget spreadsheet model.

This model holds two classes where data is to be inserted by end users: *i)* **Year**, with a default value of 2010, for the budget to accommodate multi-year information and *ii)* **Category**, for assigning a label to each expense. The actual spreadsheet may hold several repetitions of any of these elements, as indicated by the ellipsis. For each expense we record its quantity and its cost (with 0 as default value), and we calculate the total amount associated with it. Finally, (simple) summation formulas are used to calculate the global amount spent per year (cell D5), the amount spent per expense type in all years (cell F3) and the total amount spent in all years (cell F5) are also calculated.

Erwig *et al.* not only introduced *ClassSheets*, but they also developed a tool - the Gencel tool [12] - that given a *ClassSheet* model generates an instance (*i.e.* a concrete spreadsheet) that conforms to the model. Figure 2 presents a possible spreadsheet as generated by Gencel given the *ClassSheet* shown in Figure 1 (and after the end user manually introduced soma data). In this particular case, the spreadsheet is used to record the annual budget for travel and accommodation expenses of an institution.

	A	B	C	D	E	F	G	H	I
1	**Budget**	**Year**	2010		**Year**	2011		...	
2	**Category**	Qnty	Cost	Total	Qnty	Cost	Total	...	Total
3	Travel	2	320	640	7	420	2940	...	3580
4	Accommodation	5	140	700	8	185	1480	...	2180
5	...	...	...	...	...	...	...	...	...
6				1340			4420	...	5760

Fig. 2: Budget spreadsheet instance.

Since the spreadsheet is generated using all the information in the model, it is able of providing some correctness guarantees: formulas are kept consistent while new years are added, for example. Note also that, throughout the years, cost and quantity are registered for two types of expenses: travel and accommodation, and that formulas are used to calculate the total expense amounts.

Spreadsheet Evolution. At the end of 2011, the spreadsheet of Figure 2 needs to be modified to accommodate 2012 data. Most spreadsheet users would typically take four steps to perform this task: *i)* insert three new columns; *ii)* copy all the labels ("Year", "Qnty", "Cost" and "Total"); *iii)* copy all the formulas (to compute the total amount spent per expense type in 2012, and the total expense for that same year) and *iv)* update all the necessary formulas in the last column to account for the new year information. More experienced users would possibly shortcut these steps by copy-inserting, for example, the 3-column block of 2011 and changing the label "2011" to "2012" in the copied block. Still, the range of the multi-year totals must be manually extended to include the new year information. In any (combination) of these situations, a conceptually unitary edition, *add year*, needs to be executed via an error-prone combination of steps.

This is precisely the main advantage of model-driven spreadsheet development: it is possible to provide unitary transformations such as the addition of

class instances (e.g., a year or a category) as one-step procedures, while all the structural impacts of such transformations are handled automatically (e.g., the involved formulas being automatically updated). This advantage is exploited to its maximum when the model and the instance are part of the same spreadsheet development environment, as it was proposed for OpenOffice in [5].[4] Besides automation, it is also guaranteed that this type of instance level operations does not affect the model-instance conformity binding.

There are, however, several situations in which the user prefers to change a spreadsheet instance (or a particular model) in such a way that, after the edit, it will no longer conform to the previously defined model (or the respective instance). For example, if the user wants to add a column containing a possible expense discount for a particular year only, this is a trivial operation to perform at the data level which is actually not simple to perform at the model level. Therefore choosing to evolve the original spreadsheet, we may obtain the one given in Figure 3, which no longer conforms to the model of Figure 1 (the discount was added in column K).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	**Budget**	**Year**	2010		**Year**	2011		...	**Year**	2012			...	
2	**Category**	Qnty	Cost	Total	Qnty	Cost	Total	...	Qnty	Cost	Discount	Total	...	Total
3	Travel	2	320	640	7	420	2940	...	1	420	0.1	378	...	3958
4	Accommodation	5	140	700	8	185	1480	...	8	185	0	1480	...	3660
5	...	...	...	...	...	...	...	...	...	...		...	...	...
6				1340			4420	...				1858	...	7618

Fig. 3: Budget spreadsheet instance with an extra column.

One possible model evolution that regains conformity is shown in Figure 4: a class for years with 3 columns is kept, while a new class for years with an extra discount column is introduced.

	A	B	C	D	E	F	G	H	I	J	K
1	**Budget**	**Year**	year=2010		...	**Year**	year=2010			...	
2	**Category**	Qnty	Cost	Total	...	Qnty	Cost	Discount	Total	...	Total
3	name=""	qnty=0	cost=0	total=qnty*cost	...	qnty=0	cost=0	disc=0	total=qnty*cost*(1-disc)	...	total=SUM(total)
4	...	...	...	...	...	...	...		...	...	...
5				total=SUM(total)	...				total=SUM(total)	...	total=SUM(year.total)

Fig. 4: Budget spreadsheet model with an extra column.

In the remainder of this paper, we study the evolution of spreadsheet models and instances in a systematic way. As a result of our work, we present a bidirectional framework that maintains the consistency between a model and its instance. By being bidirectional, it supports either manually evolving the spreadsheet instances, as we have described in this section, or editing the model instead. In any case, the correlated artifact is automatically co-evolved, so that their conformity relationship is always preserved.

3 Spreadsheet Evolution Environment

This section presents a bidirectional spreadsheet evolution environment. This environment combines the following techniques:

[4] Actually, Figure 1 and Figure 2 present a *ClassSheet* model and a spreadsheet instance as defined in the embedding of *ClassSheets* in spreadsheets [5].

– firstly, we embed *ClassSheet* models in a spreadsheet system. Since the visual representation of *ClassSheets* very much resembles spreadsheets themselves, we have followed the traditional embedding of a domain specific language (*ClassSheets*) in a general purpose programming language (spreadsheet system). In this way, we can interact with both the models and the instances in the same environment as described in [4, 7].
– secondly, we construct a framework of bidirectional transformations for *ClassSheets* and spreadsheet instances, so that a change in an artifact is automatically reflected to its correlated one. This framework provides the usual end-user operations on spreadsheets like adding a column/row or deleting a column/row, for example. These operations can be realized in either a model or an instance, and the framework guarantees the automatic synchronization of the two. This bidirectional engine, that we describe in detail in the next section, is defined in the functional programming language HASKELL [18].
– finally, we extend the widely used spreadsheet system Calc, which is part of OpenOffice, in order to provide a bidirectional model-driven environment to end-users. Evolution steps at the model and the instance level are available as new buttons that extend the functionalities originally built-in the system. A script in OpenOffice Basic was developed to interpret the evolution steps and to make the bridge with the HASKELL framework. An OpenOffice extension is available at the SSaaPP project web page: `http://ssaapp.di.uminho.pt/`.

In Figure 5, we present an overview of the bidirectional spreadsheet environment that we propose. On the left, the embedded *ClassSheet* is presented in a Model worksheet while the data instance that conforms to it is given on the right, in a Data worksheet. Both worksheets contain buttons that perform the evolution steps at the model and instance levels.

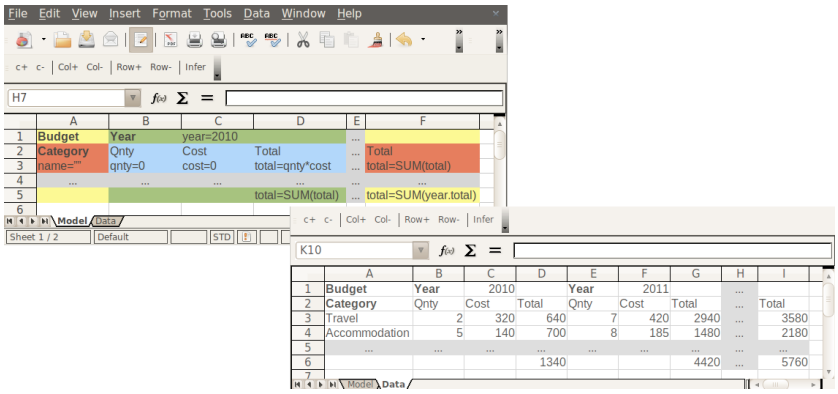

Fig. 5: A bidirectional model-driven environment for the budget spreadsheet.

Every time a (model or instance) spreadsheet evolution button is pressed, the system responds automatically. Indeed, it was built to propagate one update

at a time and to be highly interactive by immediately giving feedback to users. A global overview of the system's architecture is given in Figure 6.

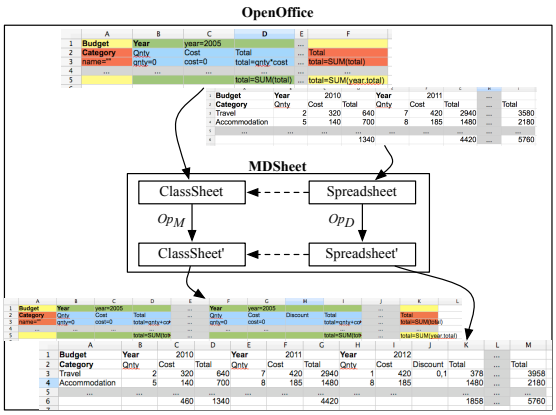

Fig. 6: Architecture of the MDSheet addon for OpenOffice.

Also, our bidirectional spreadsheet engine makes some natural assumptions on the models it is able to manipulate and restricts the number of operations that are allowed on the instance side.

Model evolution steps. On the model side, we assume the *ClassSheet* on which an editing operation is going to be performed is well-formed. By being well-formed we mean a model respecting the original definition of *ClassSheets* [10], where all references made in all formulas are correctly bound. Also, a concrete model evolution operation is only actually synchronized if it can be applied to the initial model and the evolved model remains well-formed; otherwise, the operation is rejected and the original model remains unchanged. The intuition behind this is that operations which cannot be applied to one side should not be put to the other side, but rejected by the environment. A similar postulate is made in [16].

For example, an operation that would produce an ill-formed *ClassSheet* model occurs when removing the cost column in Figure 1. In this case, the formula that computes the total expense amount per year would point to a non-existing cost value. This is precisely one of the cases that our system rejects.

Data evolution steps. The success of spreadsheets comes from the simplicity and freedom that they provide to end-users. This freedom, however, is also one of the main causes of errors in spreadsheets. In our evolution environment, we need to restrict the number of operations that a user may request. The reason for this is the following: for any supported operation, we guarantee that there exists a concrete model that the evolved instance will conform to; and for this we need to reduce the huge number of operations that is available in a spreadsheet system such as Calc, so that we can ensure model-instance conformity at all times.

As an example of an operation on an instance that we are not able to propagate to its conforming model is the random addition of data rows and columns. Indeed, if such an edit disrespects the structure of the original spreadsheet, we will often be unable to infer a new model to which the data conforms. Therefore, the operations (such as addColumn) that may affect the structure of a spreadsheet instance need to be performed explicitly using the corresponding button of our framework. The remaining editing operations are performed as usually.

4 The MDSheet Framework

In this section, we present our framework for bidirectional transformations of spreadsheets. The implementation is done using the functional programming language HASKELL and we will use some of its notation to introduce the main components of our framework. After defining the data types that encode models and instances, we present two distinct sets of operations over models and instances. We then encode a bidirectional system providing two unidirectional transformations *to* and *from* that map operations on models into operations on instances, and operations on instances to operations on models, respectively. Figure 7 illustrates our bidirectional system:

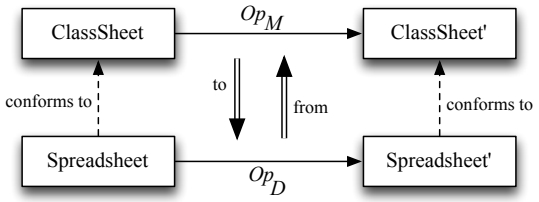

Fig. 7: Diagram of our spreadsheet bidirectional transformational system.

Given a spreadsheet that conforms to a *ClassSheet* model, the user can evolve the model through an operation of the set Op_M, or the instance through an operation of the set Op_D. The performed operation on the model (data) is then transformed into the corresponding operation on the data (model) using the *to* and *from* transformations, respectively. A new model and data are obtained with the new data conforming to the new model.

4.1 Specification of Spreadsheets and Models

The operations defined in the next sections operate on two distinct, but similar, data types. The first data type, named *Model*, is used to store information about a model. It includes the list of classes that form the model and a grid (a matrix) that contains the definition of the *ClassSheet* cells. The second type, *Data*, is used to store information about model instances, *i.e.*, the spreadsheet data. It

also stores a list of classes and a matrix with the cell contents. The definition of these data types is as follows:

```
data Model = Model { classes :: [ModelClass], grid :: Grid }
data Data  = Data  { classes :: [DataClass],  grid :: Grid }
```

The difference between the two data types lies in the kind of classes used. Models define classes with repetitions, but do not have repetitions *per se*. However, the data can have several instances, and that information is stored within the class. For a class, we store its name, the position of its top-left and bottom-right points (respectively *tl* and *br* in the data structure below) and the kind of expansion of the class.

```
data ModelClass = ModelClass {
    classname :: String
  , tl        :: (Int, Int)
  , br        :: (Int, Int)
  , expansion :: Expansion () }
```

```
data DataClass = DataClass {
    classname :: String
  , tl        :: (Int, Int)
  , br        :: (Int, Int)
  , expansion :: Expansion Int }
```

In *DataClass*, the number of instances is stored in the *expansion* field. In *ModelClass*, the *expansion* is used to indicate the kind of expansion of a class. It is possible to store the number of instances for horizontal and for vertical expansions, and for classes that expand both horizontally and vertically. It is also possible to represent static classes (*i.e.*, that do not expand).

Having introduced data types for *ClassSheet* models and spreadsheet instances, we may now define operations on them. The next two sections present such operations.

4.2 Operations on Spreadsheet Instances

The first step in the design of our transformational system is to define the operations available to evolve the spreadsheets. The grammar shown next defines the operations the MDSheet framework offers.

```
data Op_D : Data → Data =
    addColumn_D  Where Index                  -- add a column
  | delColumn_D        Index                  -- delete a column
  | addRow_D     Where Index                  -- add a row
  | delRow_D           Index                  -- delete a row
  | AddColumn_D  Where Index                  -- add a column to all instances
  | DelColumn_D        Index                  -- delete a column from all instances
  | AddRow_D     Where Index                  -- add a row to all instances
  | DelRow_D           Index                  -- delete a row from all instances
  | replicate_D  ClassName Direction Int Int  -- replicate a class
  | addInstance_D ClassName Direction Model   -- add a class instance
  | setLabel_D   (Index, Index) Label         -- set a label
  | setValue_D   (Index, Index) Value         -- set a cell value
  | SetLabel_D   (Index, Index) Label         -- set a label in all instances
  | SetValue_D   (Index, Index) Value         -- set a cell value in all instances
```

To each entry in the grammar corresponds a particular function with the same arguments. The application of an update $op_D : Op_D$ to a data instance $d : Data$ is denoted by $op_D\ d : Data$.

The first operation, $addColumn_D$, adds a column in a particular place in the spreadsheet. The *Where* argument specifies the relative location (*Before* or *After*) and the given *Index* defines the position where to insert the new column. This solves ambiguous situations, like for example when inserting a column between two columns from distinct classes. The behavior of $addColumn_D$ is illustrated in Figure 8.

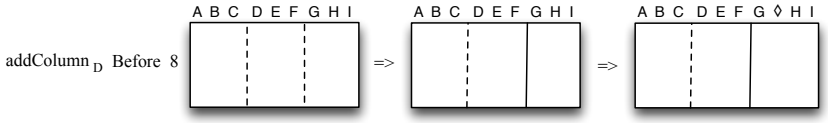

Fig. 8: Application of the data operation $addColumn_D$.

In an analogous way, the second operation, $delColumn_D$, deletes a given column of the spreadsheet. The operations $addRow_D$ and $delRow_D$ behave as $addColumn_D$ and $delColumn_D$, but work on rows instead of on columns. An operation in all similar to $addColumn_D$ ($delColumn_D$, $addRow_D$ and $delRow_D$) is $AddColumn_D$ ($DelColumn_D$, $AddRow_D$ and $DelRow_D$). This operation is in fact a mapping of $addColumn_D$ over all instances of a class: it adds a column to each instance of an expandable class. The operation $replicate_D$ allows to replicate (or duplicate) a class, with the two last integer arguments being the number of instances of the class provided as first argument and the number of the instance to replicate, respectively. This operation will be useful for our bidirectional transformation functions, and will be explained in more detail later. The operation $addInstance_D$ performs a more complex evolution step: it adds a new instance of a given class to the spreadsheet. For the example illustrated in Figure 2, it could be used to add a new instance of the year class. The operations described so far work on sets of cells (*e.g.* rows and columns), whereas the last two Op_D operations, $setLabel_D$ and $setValue_D$, work on a single cell. The former allows to set a new label to a given cell while the latter allows to update the value of a cell. Versions of these last two operations that operate on all instances are also available: $SetLabel_D$ and $SetValue_D$, respectively.

When adding a single column to a particular instance with several columns, the chosen instance becomes different than the others. Therefore, this operation is based on two steps: firstly, the chosen instance is separated from the others (note that the second dashed line becomes a continuous line); secondly, a new column, indicated by $\Diamond$, is inserted in the specified index. This operation can be used to evolve the data of the budget example as suggested at the end of Section 2 (and illustrated in Figure 3).

4.3 Operations on Models

In this section we present the operations that allow transformations on the model side. The grammar shown next declares the existing operations:

```
data Op_M : Model → Model =
    addColumn_M   Where Index                          -- add a new column
  | delColumn_M          Index                         -- delete a column
  | addRow_M      Where Index                          -- add a new row
  | delRow_M             Index                         -- delete a row
  | setLabel_M    (Index, Index) Label                 -- set a label
  | setFormula_M  (Index, Index) Formula               -- set a formula
  | replicate_M   ClassName Direction Int Int          -- replicate a class
  | addClass_M    ClassName (Index, Index) (Index, Index) -- add a static class
  | addClassExp_M ClassName Direction (Index, Index) (Index, Index)
                                                       -- add an expandable class
```

As it occurred with Op_D for instances, the Op_M grammar represents the functions operating on models. The application of an update $op_M : Op_M$ to a model $m : Model$ is denoted by $op_M\ m : Model$.

The first five operations are analogous to the data operations with the same name. New operations include $setFormula_M$ which allows to define a formula on a particular cell. On the model side, a formula may be represented by an empty cell, by a default plain value (*e.g.*, an integer or a date) or by a function application (*e.g.*, cell F5 in Figure 2 is defined as the result of `SUM(Year.total)`). The operation $replicate_M$ allows to replicate (or duplicate) a class. This will be useful for our bidirectional transformation functions. The last two operations allow the addition of a new class to a model: $addClass_M$ adds a new static (non-expandable) class and $addClassExp_M$ creates a new expandable class. The *Direction* parameter specifies if it expands horizontally or vertically.

To explain the operations on models, we present in Figure 9 an illustration of the execution of the composition of two operations: firstly, we execute an addition of a row, $addRow_M$ (where the new row is denoted by ◊); secondly, we add a new expandable class ($addClassExp_M$) constituted by columns B and C (denoted by the blue rectangle and the grey column labeled with the ellipsis).

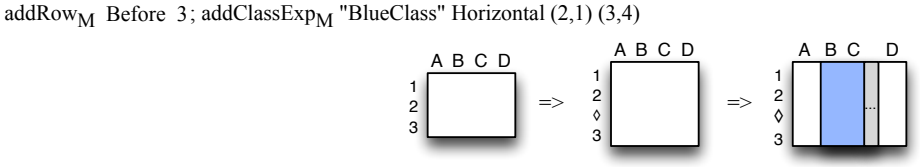

Fig. 9: Application of the model operations $addRow_M$ and $addClassExp_M$.

The first operation, $addRow_M$, adds a new row to the model between rows 2 and 3. The second operation, $addClassExp_M$, adds a new class to the model. As

a first argument, this operation receives the name of the class, which is "BlueClass" in this case. Its second argument specifies if the class expands vertically or horizontally (the latter in this case). The next two arguments represent the upper-left and the right-bottom indexes limiting the class, respectively. The last argument is the model itself.

These operations allow to evolve models like the one presented in Figure 1. However, the problem suggested in Figures 3 and 4, where the data evolves as a result of an end-user operation and the model automatically co-evolves, is not yet handled by transformations we presented so far. In the next section, we give a bidirectional setting where co-evolution is automatic.

4.4 Bidirectional Transformation Functions

In this section, we present the bidirectional core of our spreadsheet transformation framework. In our context, a bidirectional transformation is defined as a pair of unidirectional transformations with the following signature:

$$\begin{aligned} &to &&: Model \times Op_M \rightarrow Op_D^\star \\ &from &&: Data \times Op_D \rightarrow Op_M^\star \end{aligned}$$

Since these transformations are used in an online setting, in which the system reacts immediately to each user modification, the general scheme of our transformations is to transform a single update into a sequence of updates. That said, the forward transformation *to* propagates an operation on models to a sequence of operations on underlying spreadsheets, and the backward transformation *from* propagates an operation on spreadsheets to a sequence of operations on overlying models. We denote a sequence of operations on models $op^\star$ as being either the empty sequence $\emptyset$, an unary operation op, or a sequence of operations $op_1^\star; op_2^\star$. Our transformations take an additional model or instance to which the original modifications are applied. This is necessary because most of the operations calculate the indexes of the cells to be updated based on the operation itself, but also based on the previous model or spreadsheets, depending on the kind of operation.

We now instantiate the *to* and *from* transformations for the operations on models and instances defined in the previous two sections. We start by presenting the *to* transformation: it receives an operation on models and returns a sequence of operations on data. Although, at least currently, the translation of model operations returns an empty or singleton sequence of data operations, we keep its signature consistent with *from* to facilitate its understanding and evidence the symmetry of the framework.

$$\begin{aligned} &to : Op_M \rightarrow Op_D^\star \\ &to\ (addColumn_M \quad w\ i \quad) = AddColumn_D\ w\ (columnIndex_D\ i) \\ &to\ (delColumn_M \quad w\ i \quad) = DelColumn_D \quad (columnIndex_D\ i) \\ &to\ (addRow_M \quad w\ i \quad) = AddRow_D \quad w\ (rowIndex_D\ i) \\ &to\ (delRow_M \quad w\ i \quad) = DelRow_D \quad (rowIndex_D\ i) \\ &to\ (setLabel_M \quad (i,j)\ l \quad) = SetLabel_D\ (position_D\ (i,j))\ l \\ &to\ (setFormula_M \quad (i,j)\ f \quad) = SetValue_D\ (position_D\ (i,j))\ f \end{aligned}$$

$$
\begin{aligned}
&to\ (replicate_M \quad cn\ dir\ n\ inst) = replicate_D\ dir\ cn\ n\ inst\\
&to\ (addClass_M \quad cn \quad p_1\ p_2\ ) = \emptyset\\
&to\ (addClassExp_M\ cn\ dir\ p_1\ p_2\ ) = \emptyset
\end{aligned}
$$

The first five model operations have a direct data transformation, that is, a transformation with the same name which does the analogous operation. An interesting transformation is perfomed by $replicate_M$: it duplicates a given class, but not the data. Instead, a new empty instance is added to the newly created class. Another interesting case is the transformation of the model operation $addClass_M$. This model transformation does not have any impact on data instances. Thus, *to* returns an empty set of data transformations. In fact, the same happens with the $addClassExp_M$ operation.

We now present the *from* transformation, which maps data operations into model operations:

$$
\begin{aligned}
&from : Op_D \rightarrow Op^{\star}_M\\
&from\ (addColumn_D\ w\ i) =\\
&\quad replicate_M\ className\ Horizontal\ classInstances\ instanceIndex_M\\
&\quad ; addColumn_M\ w\ columnOffsetIndex_M\\
&from\ (delColumn_D\ i) =\\
&\quad replicate_M\ className\ Horizontal\ classInstances\ instanceIndex_M\\
&\quad ; delColumn_M\ columnOffsetIndex_M\\
&from\ (addRow_D\ w\ i) =\\
&\quad replicate_M\ className\ Vertical\ classInstances\ rowIndex_M\\
&\quad ; addRow_M\ w\ rowOffsetIndex_M\\
&from\ (delRow_D\ i) =\\
&\quad replicate_M\ className\ Vertical\ classInstances\ rowIndex_M\\
&\quad ; delRow_M\ rowOffsetIndex_M\\
&from\ (setLabel_D\ (i,j)\ l) =\\
&\quad replicate_M\ className\ Horizontal\ classInstances\ columnIndex_M\\
&\quad ; replicate_M\ className\ Vertical \quad classInstances\ rowIndex_M\\
&\quad ; setLabel_M\ positionOffset_M\ l\\
&from\ (setValue_D\ (i,j)\ l \quad) = \emptyset\\
&from\ (addInstance_D\ cn\ dir\ m) = \emptyset
\end{aligned}
$$

The transformations in this case are more complex than the model-to-data ones. In fact, most of them produce a sequence of model operations. For instance, the first transformation ($addColumn_D$) results in the replication of a class followed by the addition of a new column. The argument *classInstances* is actually a function that calculates the number of data class instances based on the data to be evolved. On the other hand, the operation to set a value of a particular cell, $setValue_D$, does not have any impact on the model. The same happens to the operation $addInstance_D$ which adds a new instance of an expandable class. The definition of *from* for the (non-empty) data operations in the range of *to* (*e.g.*, $AddColumn_D, DelRow_D$) is simply the inverse of *to*.

4.5 Bidirectional Transformation Properties

Since the aim of our bidirectional transformations is to restore the conformity between instances and models, a basic requirement is that they satisfy *correct-*

ness [22] properties entailing that propagating edits on spreadsheets or on models leads to consistent states:

$$\frac{d :: m}{(to\ m\ op_M)\ d :: op_M\ m}\ to\text{-Correct} \qquad \frac{d :: m}{op_D\ d :: (from\ d\ op_D)\ m}\ from\text{-Correct}$$

Here, we say that a data instance d conforms to a model m if $d :: m$, for a binary consistency relation $(::) \subseteq Data \times Model$.

As most interesting bidirectional transformation scenarios, spreadsheet instances and models are not in bijective correspondence, since multiple spreadsheet instances may conform to the same model, or vice versa. Another bidirectional property, *hippocraticness* [22], postulates that transformations are not allowed to modify already consistent pairs, as defined for *from*:

$$\frac{d :: m \qquad op_D\ d :: m}{from\ d\ op_D = \emptyset}\ from\text{-Hippocratic}$$

Reading the above law, if an operation on data op_D preserves conformity with the existing model, then *from* produces an empty sequence of operations on models $\emptyset$. Operationally, such a property is desired in our framework. For example, if a user adds a new instance to an expandable class, preserving conformity, the original model is expected to be preserved because it still reflects the structure of the data. However, hippocracticness for *to* is deemed too strong because even if the updated model is still consistent with the old data, we still want to update the data to reflect a change in the structure, making a better match with the model [8]. For example, if the user adds a column to the model, the intuition is to insert also a new column to the data, even if the old data remains consistent.

Usually, bidirectional transformations are required to satisfy "round-tripping" laws that ensure a certain degree of *invertibility* [9, 13]. In our application, spreadsheet instances refine spreadsheet models, such that we can undo the translation of an operation on models with an application of *from* (except when the operation on models only concerns layout, such as $addClass_M$, and is not reflexible on the data):

$$\frac{to\ m\ op_M = op_D^\star \qquad op_D^\star \neq \emptyset \qquad d :: m}{from^\star\ d\ op_D^\star = op_M}\ to\text{-Invertible}$$

However, the reverse implication is not true. For example, the transformation steps $from\ addColumn_D = replicate_M; addColumn_M$ and $to^\star\ (replicate_M; addColumn_M) = replicate_D; AddColumn_D$ do not give equal data operation.

Since an operation on data may issue a sequence of operations on models, we introduce the transformation $to^\star$ which applies *to* to a sequence of operations:

$$\frac{}{to^\star\ \emptyset = \emptyset} \qquad \frac{}{to^\star\ op_M = to\ op_M} \qquad \frac{to^\star\ op_{M1}^\star = op_{M1}^\star \qquad to^\star\ op_{M2}^\star = op_{M2}^\star}{to^\star\ (op_{M1}^\star; op_{M2}^\star) = to^\star\ op_{M1}^\star; to^\star\ op_{M2}^\star}$$

A dual definition can be given for $from^{\star}$. As common for incremental transformations (that translate edits rather than whole states), our sequential transformations naturally satisfy an *history ignorance* [8] property meaning that the translation of consecutive updates does not depend on the update history.

5 Related Work

Our bidirectional approach is inspired in the state-based bidirectional framework of (constraint) maintainers [20, 22], where (correct and hippocratic) forward and backward transformations propagate modifications from source to target models, and vice-versa, while preserving a consistency relation that establishes a relationship between them. However, our formulation is closer to operation-based symmetric bidirectional frameworks [9, 16]. The framework of symmetric delta lenses from [9] generalizes maintainers to transformations that operate over deltas as high-level representations of updates. Like [16], our transformations carry a more operational feeling as they transform the actual operations on data and models. Our bidirectional transformations also satisfy a similar totality law guaranteeing that if an operation does not fail on the initiating side, then the transformed sequence of operations also succeeds. For example, given a consistent state $d :: m$, propagating a data operation over d always generates operations on *ClassSheet* models that can be applied to the original model m.

A group of researchers from Tokyo developed a series of bidirectional approaches for the interactive development of XML documents [17, 23]. Similarly to our OpenOffice assisted environment, they assume an online setting where the editor reacts immediately to one operation at a time. In their setting, instead of preserving an explicit consistency relation, transformations obey one-and-a-half round-tripping laws (in the style of $to \circ from \circ to = to$) to ensure that after each modification the editor converges into a consistent state, *i.e.*, a further transformation does not alter the related documents.

The coupled evolution of metamodels (let it be grammars, schemas, formats, etc) and conforming models is a typical problem in MDE. Works such as [15, 19] assess the degree of automation of metamodel-model evolution scenarios by studying categories of metamodel modifications that are model-independent or support the co-evolution of underlying models which need to be transformed in order to become conforming to an updated version of their original metamodel. Existing tools for automated coupled metamodel-model evolution may either require users to specify sequences of simple that describe how to evolve a source metamodel into a new version [24], or assume that a new metamodel is provided externally so that the system must user model difference approaches to identify the concrete metamodel changes [3]. Both [24] and [3] support typical metamodel operations such as renaming, addition and deletion and many others to manipulate particular object-oriented features. The *to* transformation proposed in this paper tackles an instance of this problem (concerned with translating a single modification at a time), with metamodels as classsheets and models

as spreadsheets. In our bidirectional setting, the *from* transformation tackles another dual but less common coupled model-metamodel evolution problem.

In [4–7], the authors introduced tools to transform spreadsheets into relational databases, and more generically, to evolve a model and automatically co-evolve the underlying data. This work, however, has some limitations: first, it does not allow users to perform non-refinement evolutions, i.e., it is not possible to remove data from spreadsheets. In our work we created a more general setting where all kinds of evolutions are possible, including the deletion of data. Second, it is not possible to evolve the structure of the spreadsheet through changes to the data, i.e., it is only possible to edit the data in such a way that it always conforms to the model. We have solved this problem by allowing users to change the data and infer a new model whenever necessary. Third, the previous work propagates modified states into new states. This work propagates editing operations themselves, and thus allows for more efficient incremental transformations.

The first approach to deliver model-driven engineering to spreadsheet users, *Gencel* [11], generates a new spreadsheet respecting a previously defined model. In this approach, however, 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 is, the co-evolution of the model (instance) and its instance (model) is not possible, unless it is done by hand, which is very error prone and thus not desirable. In our work we present a solution for this problem.

6 Conclusions and Future Work

In this paper we have presented a bidirectional model-driven spreadsheet environment. We constructed a bidirectional framework defining usual end-user operations on spreadsheet data and on *ClassSheet* models that always guarantee synchronization after an evolution step at the data or model level. We have created an extension to the OpenOffice Calc spreadsheet system so that it offers a model-driven software development environment. The developed spreadsheet evolution environment allows: the generation of a spreadsheet instance from a *ClassSheet* model; the evolution of the model and the automatic co-evolution of the data; the evolution of the data and the automatic co-evolution of the model.

The techniques we present propose the first bidirectional setting for the evolution of spreadsheet models and instances. Our research efforts, however, have thus far considered standalone, non-concurrent spreadsheet development only. In a computing world that is growingly distributed, developing spreadsheets is often performed in a collaborative way, by many actors. As part of our plans for future research, we have already engaged in trying to extend our work in this paper to a distributed environment.

Although there exists some empirical evidence that an approach to spreadsheet development based on models can sometimes be effective in practice [2], the global environment envisioned in this paper still lacks a concrete empirical analysis. In this line, a study with real spreadsheet users is under preparation.

References

1. Abraham, R., Erwig, M., Kollmansberger, S., Seifert, E.: Visual specifications of correct spreadsheets. In: VL/HCC. pp. 189–196. IEEE Computer Society (2005)
2. Beckwith, L., Cunha, J., Fernandes, J.P., Saraiva, J.: End-users productivity in model-based spreadsheets: An empirical study. In: IS-EUD. pp. 282–288 (2011)
3. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: EDOC. pp. 222–231. IEEE CS (2008)
4. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: MDSheet: A Framework for Model-driven Spreadsheet Engineering. In: ICSE. pp. 1412–1415. ACM (2012)
5. Cunha, J., Mendes, J., Fernandes, J.P., Saraiva, J.: Embedding and evolution of spreadsheet models in spreadsheet systems. In: VL/HCC '11. pp. 179–186. IEEE
6. Cunha, J., Saraiva, J., Visser, J.: From spreadsheets to relational databases and back. In: PEPM. pp. 179–188. ACM, New York, USA (2009)
7. Cunha, J., Visser, J., Alves, T., Saraiva, J.: Type-safe evolution of spreadsheets. In: FASE. pp. 186–201. Springer-Verlag, Berlin, Heidelberg (2011)
8. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: MoDELS 2008. pp. 21–36. Springer (2008)
9. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: the symmetric case. In: MODELS. pp. 304–318. Springer-Verlag, Berlin, Heidelberg (2011)
10. Engels, G., Erwig, M.: ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In: ASE. pp. 124–133. ACM (2005)
11. Erwig, M., Abraham, R., Cooperstein, I., Kollmansberger, S.: Automatic generation and maintenance of correct spreadsheets. In: ICSE. pp. 136–145. ACM (2005)
12. Erwig, M., Abraham, R., Kollmansberger, S., Cooperstein, I.: Gencel: a program generator for correct spreadsheets. J. Funct. Program 16(3), 293–325 (2006)
13. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In: POPL. pp. 233–246. ACM (2005)
14. Hermans, F., Pinzger, M., van Deursen, A.: Automatically extracting class diagrams from spreadsheets. In: ECOOP. pp. 52–75. Springer-Verlag (2010)
15. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In: MoDELS, LNCS, vol. 5301, pp. 645–659. Springer (2008)
16. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: POPL. to appear (2012)
17. Hu, Z., Mu, S.C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. HOSC 21(1–2), 89–118 (2008)
18. Jones, S.P., Hughes, J., Augustsson, L., et al.: Report on the programming language haskell 98. Tech. rep. (February 1999)
19. Lämmel, R., Lohmann, W.: Format Evolution. In: RETIS 2001. vol. 155, pp. 113–134. OCG (2001)
20. Meertens, L.: Designing constraint maintainers for user interaction (1998), manuscript available at `http://www.kestrel.edu/home/people/meertens`
21. Panko, R.: Spreadsheet errors: What we know. what we think we can do. EuSpRIG (2000)
22. Stevens, P.: Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In: MoDELS 2007, LNCS, vol. 4735, pp. 1–15. Springer (2007)
23. Takeichi, M.: Configuring bidirectional programs with functions. In: IFL (2009)
24. Vermolen, S., Visser, E.: Heterogeneous coupled evolution of software languages. In: MoDELS 2008, LNCS, vol. 5301, pp. 630–644. Springer (2008)