Embedding, Evolution, and Validation of Model-Driven Spreadsheets

Jácome Cunha, João Paulo Fernandes,
Jorge Mendes, João Saraiva
{jacome,jpaulo,jorgemendes,jas}@di.uminho.pt

Techn. Report TR-HASLab:01:2014

Jun. 2014

SSaaPP – SpreadSheets as a Programming Paradigm
(Project FCOMP-01-0124- FEDER-010048)

HASLab - High-Assurance Software Laboratory
Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://haslab.di.uminho.pt

TR-HASLab:01:2014
Embedding, Evolution, and Validation of Model-Driven Spreadsheets
by Jácome Cunha, João Paulo Fernandes,
Jorge Mendes, João Saraiva

Abstract

This paper proposes and validates a model-driven software engineering technique for spreadsheets. The technique that we envision builds on the embedding of spreadsheet models under a widely used spreadsheet system, so that models and their conforming instances are developed under the same environment. In practice, this convenient environment enhances evolution steps at the model level while the corresponding instance is automatically co-evolved. Finally, we have designed and conducted an empirical study with human users in order to assess our technique in production environments. The results of this study are promising and suggest that productivity gains are realizable under our model-driven spreadsheet development setting.

1 Introduction

The use of abstract models to reason about concrete artifacts has been successfully employed in science and in engineering. In fact, there are many fields for which model-driven engineering is the default, uncontested approach to follow: it is a reasonable assumption that, excluding financial or cultural limitations, no private house, let alone a bridge or a skyscraper, should be built before its model has been created and thoroughly analyzed and evolved.

Being itself a considerably more recent scientific field, not many decades have passed since software engineering has seriously considered the use of models. In our work, we have focused our attention on model-driven approaches to spreadsheet software engineering. Spreadsheets are a relevant research topic, as they play a pivotal role in modern society. Indeed, they are inherently multi-purpose and widely used both by individuals to cope with simple needs as well as large companies as integrators of complex systems and as support for informing business decisions [1]. Also, their popularity is still growing, with an almost impossible to estimate but staggering number of spreadsheets created every year. Spreadsheet popularity is due to characteristics such as their low entry barrier, their availability on almost any computer, and their simple visual interface. In fact, being a conventional language that is understood by both professional programmers and end users [2], spreadsheets are often used as bridges between these two communities which often face communication problems. Ultimately, spreadsheets seem to hit the sweet spot between flexibility and expressiveness.

Spreadsheets have probably passed the point of no return in terms of importance: it is estimated that 95% of all U.S. firms use them for financial reporting [3], 90% of all analysts in industry perform calculations in spreadsheets [3], and 50% of all spreadsheets are the basis for decisions [1]. This importance, however, has not been achieved together with effective mechanisms for error prevention, as shown by several studies [4, 5]. This claim is also supported by the long list of real problems that were blamed on spreadsheets, which has been documented and made available at the *European Spreadsheet Risk Interest Group (EuSpRIG)* web site[1].

In an attempt to overcome the issue of spreadsheet errors using model-driven approaches, several techniques have been proposed, namely the creation of spreadsheet templates [6], the definition of *ClassSheet* [7] models and the inference of class diagrams from spreadsheets [8]. These proposals guarantee that users may safely perform particular editing steps on 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 guaranteeing the consistency of the spreadsheet with respect to the underlying model.

Despite 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 with both the spreadsheet domain (business logic) and model-driven software development. In the particular case of the use of templates, a new tool is necessary to be learned, namely the ViTSL [9] template editor. In fact, templates are created with ViTSL, which is similar to Excel, but offers additional template specific functionality. These templates can then be loaded into the *Gencel* [10], system, which is implemented as an extension to Excel, and allows the safe, error-free editing of spreadsheets. This approach, however,

[1] This list of horror stories is available at: `http://www.eusprig.org/horror-stories.htm`

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 is, the co-evolution of the model and its instance is not possible.

The first contribution of our work is the embedding 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 worksheet we define the underlying model while another worksheet holds the actual data, such that the model and the data are kept synchronized by our framework. A summarized description of this work has been presented in [11], a description that we revise and extend in this paper, in Section 3.

The second contribution of our work builds on this coherent spreadsheet development setting where it is possible to define both the model and the data of a concrete spreadsheet. Indeed, having both artifacts under the same environment enhances the possibility of synchronizing one artifact in response to concrete evolution steps on the other. In the context of the work in this paper, we focus on describing and assessing evolution mechanisms on the model side and on the co-evolution of the data instances in response to that evolution, as originally proposed in [12]. This contribution is introduced in Section 4.

The third contribution of this paper is the design, execution and analysis of an experiment with real users we have conducted to assess the spreadsheet development framework previously described. Although we have ran other empirical studies in the past – [13, 14] – they do not evaluate the embedding and evolution setting we present in this work. Thus, the study we include in this paper intends to confirm two aspects of spreadsheet development: (1) that users spend less time to perform a series of actions using or model-driven setting; and (2) that we end up with spreadsheets having fewer errors than one would expect from the traditional spreadsheet development setting. The details of our study are presented in Section 6, where we show statistical evidences that confirm both of these hypotheses, under the conditions that we tested.

This paper is organized as follows: In Section 2, we revise the *ClassSheet* modeling language, in both its textual and visual representations, and using examples of practical interest. In Section 3, we go through the embedding of *ClassSheets* in spreadsheet systems. Section 4 presents evolution rules and how to evolve models and automatically co-evolve the corresponding instances. In Section 5 we present the tool implemented using the previously described techniques. The empirical validation of our approach to spreadsheet development is presented in Section 6. Finally, Section 7 presents related work and Section 8 concludes the paper.

2 Modeling Spreadsheets with *ClassSheets*

ClassSheets are a high-level, object-oriented formalism to specify the business logic of spreadsheets [7]. This formalism allows users to express business object structures within a spreadsheet using concepts from the UML [15].

ClassSheets define (work)sheets (s) containing classes (c) formed by blocks (b), which can be expandable, either horizontally ($c^{\rightarrow}$) or vertically ($b^{\downarrow}$). Also, classes are identified by labels (l), and a block may represent in its basic form a spreadsheet cell, or it can be

a composition of other blocks. When representing a cell, a block can contain a basic value (φ, e.g., a string or an integer) or an attribute ($a = f$), which is composed by an attribute name (a) and a value (f). Attributes can define three types of cells: (1), an input value, where a default value gives that indication, (2), a named reference to another attribute ($n.a$, where n is the name of the class and a the name of the attribute) or (3), an expression built by applying functions to a varying number of arguments given by a formula ($\varphi(f, \ldots, f)$).

ClassSheets can be represented textually, according to the grammar presented in Figure 1 and taken directly from [7], or visually as described further below.

$$
\begin{array}{lcll}
f \in Fml & ::= & \varphi \mid n.a \mid \varphi(f, \ldots, f) & (formulas) \\
b \in Block & ::= & \varphi \mid a = f \mid b|b \mid b\hat{}b & (blocks) \\
l \in Lab & ::= & h \mid v \mid .n & (class\ labels) \\
h \in Hor & ::= & \underline{n} \mid |\underline{n} & (horizontal) \\
v \in Ver & ::= & |n \mid |\underline{n} & (vertical) \\
c \in Class & ::= & l:\ b \mid l:\ b^{\downarrow} \mid c\hat{}c & (classes) \\
s \in Sheet & ::= & c \mid c^{\rightarrow} \mid s|s & (sheets)
\end{array}
$$

Figure 1: Syntax of the textual representation of *ClassSheets*.

2.1 Vertically Expandable Tables

In order to illustrate how *ClassSheets* can be used in practice we shall consider an example modeling an airline scheduling system which we adapted from [16]. 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[2] as the one presented in Figure 2a. 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's name and **Age** represents the pilot's age. Each of the subsequent rows contains data for an actual pilot.

Tables such as the one presented in Figure 2a are frequently used within spreadsheets, and it is fairly simple to create a model specifying them. In fact, Figure 2b represents a visual *ClassSheet* model for this pilot's table, whilst Figure 2c shows the textual *ClassSheet* representation. In the following paragraphs we explain such a model.

To model the labels we use a textual representation and the exact same names as in the data sheet (**Pilots**, **ID**, **Name** and **Age**). To model the actual data we abstract concrete column cell values by using a single identifier: we use the one-worded, lowercase equivalent of the corresponding column label (so, *id*, *name* and *age*). Next, a default value 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 =). Note that the last row of the model is labeled on the left hand-side with vertical ellipses. This means that it is possible for the previous block of rows to expand vertically, that is, the tables that conform to this model can have as many rows/pilots as needed. The scope of the expansion is defined by the region between the ellipsis and the black line separating rows `2` and `3`. Note that, by definition,

[2] A *table* is a block of cells separated from the others by empty columns/rows. The first columns may contain the name of the table. Usually its first/second row contains the column labels.

	A	B	C
1	**Pilots**		
2	**ID**	**Name**	**Age**
3	pl1	John	45
4	pl2	Mike	39
5	pl3	Anne	56

(a) Pilots' table.

	A	B	C
1	**Pilots**		
2	**ID**	**Name**	**Age**
3	id=""	name=""	age=0
⋮			

(b) Pilots' visual *ClassSheet* model.

Pilots : Pilots | ⊔ | ⊔ ^
Pilots : ID | Name | Age ^
Pilots : (id= "" | name= "" | age= 0)↓

(c) Pilots' textual *ClassSheet* model.

Figure 2: Pilots' example.

ClassSheets do not allow for nested expansion blocks, and thus, there is no possible ambiguity associated with this feature. The instance shown in Figure 2a has three pilots.

2.2 Horizontally Expandable Tables

In the lines of what we described in the previous section, airline companies must also store information of their airplanes. This is the purpose of table **Planes** in the spreadsheet illustrated in Figure 3a, which is organized as follows: the first column holds labels that identify each row, namely, **Planes** (labeling the table itself), **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.

The **Planes** table can be visually modeled by the illustration in Figure 3b and textually by the definition in Figure 3c. 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 one the names abstracting concrete data values: again, each cell has a name and the default value of the elements in that row (in this example, all the cells have as default values empty strings); in Figure 3b, the third column is labeled not as `C` but with ellipses meaning the information in column `B` is horizontally expandable. Note that the instance table of Figure 3a has information about three planes.

2.3 Relationship Tables

The examples used so far (the tables for pilots and planes) are useful to store the data, but another kind of table exists and can be used to relate information, being of more

	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) Planes' table.

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

(b) Planes' visual *ClassSheet* model.

$$\left(\begin{array}{|ll} \textbf{Planes}: & \text{Planes} \\ \underline{\textbf{N-Number}}: & \text{N-Number} \\ \underline{\textbf{Model}}: & \text{Model} \\ \underline{\textbf{Name}}: & \text{Name} \end{array}\right)$$

$$\left(\begin{array}{|ll} \textbf{Planes}: & \sqcup \\ \underline{\textbf{N-Number}}: & \text{n-number= ""} \\ \underline{\textbf{Model}}: & \text{model= ""} \\ \underline{\textbf{Name}}: & \text{name= ""} \end{array}\right) \rightarrow$$

(c) Planes' textual *ClassSheet* model.

Figure 3: Planes' example.

practical interest.

Having pilots and planes, we can set up a new table to store information from the flights that the pilots make with the planes. This new table is called a *relationship* table since it relates two entities, which are the pilots and the planes. A possible model for this example is presented in Figure 4, which also depicts an instance of that model.

	A	B	C	D	E	...	F
1	**Flights**	**PlanesKey**					
2		plane_key=**Planes**.n-number					
3	**PilotsKey**	Depart	Destination	Date	Hours		Total Pilot Hours
4	pilot_key=**Pilots**.ID	depart=""	destination=""	date=d	hours=0		total=SUM(hours)
⋮							
5							total=SUM(**PilotsKey**.total)

(a) Flights' visual *ClassSheet* model.

	A	B	C	D	E	F	G	H	I	J
1	**Flights**	**PlanesKey**				**PlanesKey**				
2		N2342				N341				
3	**PilotsKey**	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours	Total Pilot Hours
4	pl1	OPO	NAT	12/12/2010 – 14:00	07:00	LIS	AMS	16/12/2010 – 10:00	02:45	09:45
5	pl2	OPO	NAT	01/01/2011 – 16:00	07:00					07:00
6										16:45

(b) Flights' table.

Figure 4: Flights' table, relating Pilots and Planes.

The flights' table contains information from distinct entities. In the model (Figure 4a), there is the class **Flights** that contains all the information, including:

- information about planes (class **PlanesKey**, columns `B` to `E`), namely a reference to the planes table (cell `B2`);
- information about pilots (class **PilotsKey**, rows `3` and `4`), namely a reference to the pilots table (cell `A4`);
- information about the flights (in the range `B3:E4`), namely the depart location (cell `B4`), the destination (cell `C4`), the time of departure (cell `D4`) and the duration of the flight (cell `E4`);
- the total hours flown by each pilot (cell `F4`), and also a grand total (cell `F5`). We assume that the same pilot does not appear in two different rows. In fact, we could use *ClassSheet* extensions to ensure this [17, 18].

For the first flight stored in the data (Figure 4b), we know that the pilot has the identifier *pl1*, the plane has the n-number *N2342*, it departed from *OPO* in direction to *NAT* at 14:00 on December 12, 2010, with a duration of 7 hours.

Note that we do not show the textual representation of this part of the model because of its complexity and because it would not improve the understandability of the paper.

3 Embedding Spreadsheet Models

The *ClassSheet* language is a Domain Specific Language (DSL) to represent the business model of spreadsheet data. Furthermore, as we have seen in the previous section, the visual representation of *ClassSheets* very much resembles spreadsheets themselves. These two facts combined motivated the use of spreadsheet systems to define *ClassSheet* models [11], that is, to natively embed *ClassSheets* in a spreadsheet host system.

Therefore, we have adopted the well-known technique to embed DSLs in a host general purpose language [19], therefore exploiting the features of the host language as *native* of the domain specific language.

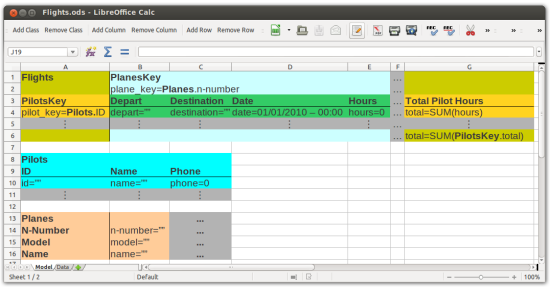
In our case, this allows both the model and the spreadsheet to be stored in the same file, and that model creation along with data editing can be handled in the same environment that users are familiar with. Also, this is achieved while not imposing on us the effort of reconstructing this environment from scratch.

The embedding of *ClassSheets* within spreadsheets is not direct, since *ClassSheets* were not meant to be embedded inside spreadsheets. Their resemblance helps, but some limitations arise due to syntactic restrictions imposed by spreadsheet host systems. Several options are available to overcome the syntactic restrictions, like writing a new spreadsheet host system from start, modifying an existing one, or adapting the *ClassSheet* visual language. The two first options are not viable to distribute Model-Driven Spreadsheet Engineering (MDSE) widely, since both require users to switch their system, which can be inconvenient. Also, to accomplish the first option would be a tremendous effort and would change the focus of the work from the embedding to building a tool.

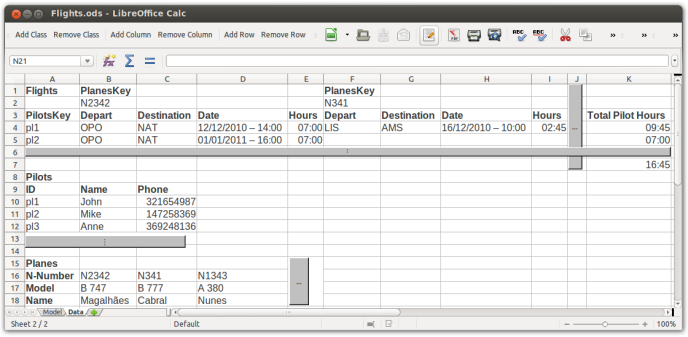
The solution adopted slightly modifies the *ClassSheet* visual language so it can be embedded in a worksheet without doing major changes on the spreadsheet host system (see Figure 5a). In these modifications, we:

1. identify horizontal (vertical) expansions using regular columns (rows) (in the *ClassSheet* language, this identification is made using specific columns/rows); such rows/columns have grey background and ellipsis as labels;
2. draw an expansion limitation black line in the spreadsheet (originally this is done between column/row letters/numbers);
3. fill classes with a background color (instead of using lines as in the original *ClassSheets*).

The last change (3) is not mandatory, but it is easier to identify the classes and, along with the first change (2), eases the identification of classes' parts. This way, users do not need to think which role the line is playing (expansion limitation or class identification).



(a) Model on the first worksheet of the spreadsheet.



(b) Data on the second worksheet of the spreadsheet.

Figure 5: Flights' spreadsheet, with an embedded model and a conforming instance.

We can use the flights' spreadsheet example to compare the differences between the original *ClassSheet* and its embedded representation:

- In the original *ClassSheet* (Figure 4a), there are two expansions: one denoted by the column between columns `E` and `F` for the horizontal expansion, and another denoted by the row between rows `4` and `5` for the vertical one. Applying change 1 to the original model will add an extra column (`F`) and an extra row (`5`) to identify the expansions in the embedding (Figure 5a).
- To define the expansion limits in the original *ClassSheet*, there are no lines between the column headers of columns `B`, `C`, `D` and `E` which makes the horizontal

expansion to use three columns and the vertical expansion to only use one row. This translates to a line between columns `A` and `B` and another line between rows `3` and `4` in the embedded *ClassSheet* as per change 2.

- To identify the classes, background colors are used (change 3), so that the class **Flights** is identified by the green[3] background, the class **PlanesKey** by the cyan background, the class **PilotsKey** by the yellow background, and the class that relates the **PlanesKey** with the **PilotsKey** by the dark green background. Moreover, the relation class (range `B3:E5`), called **PilotsKey_PlanesKey**, is colored in dark green.

In the original definition of *ClassSheet*, default values are used to represent types of value-cells. In fact, the formal definition of *ClassSheets* does not consider a pre-defined set of types. Our embedding supports the types/default values offered by the spreadsheet host system, namely: number, percent, currency, date, time, scientific, fraction, boolean value, and text. Indeed when defining the model, the user can define the default values using any value from these types. The use of this information in the instances is explained in Section 3.2.

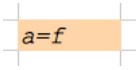
Next we present a set of rules that are used to map each element of the *ClassSheet* language to our embedding.

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

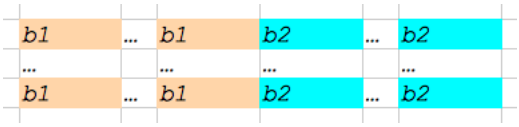
$b \in Block$

φ

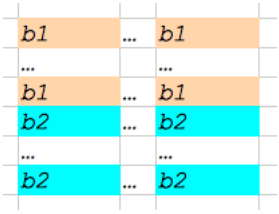
$a = f$



$b \mid b$

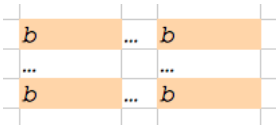


$b \hat{} b$

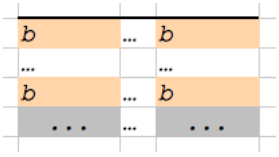


$c \in Class$

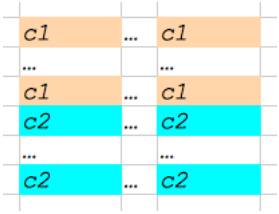
$l : b$



$l : b^{\downarrow}$

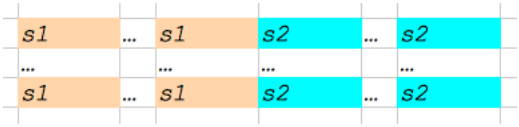


$c \hat{} c$

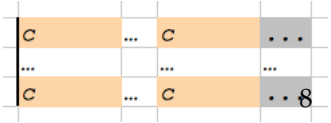


$s \in Sheet$

$s \mid s$



$c^{\rightarrow}$



The elements φ and $a = f$ are simply placed in a colored cell.

A horizontal composition of blocks ($b \mid b$) is embedded placing the first block (here represented by $b1$ labeled cells) in the corresponding cells, which are colored from the previous step. For the second block ($b2$) the same happens, although with a different color to distinguish them. As the blocks form a horizontal composition they are placed side by side in the embedded model.

A very analogous situation occurs for the vertically aligned blocks ($b \hat{} b$), but instead of begin placed side by side, they are on top of each other.

The next case is the labeled class ($l : b$). In this case, the label is not visible in the embedding; only the block is created using the previously presented rules.

A similar situation occurs for the labeled horizontally expandable class $l : b^{\downarrow}$. The block is created using the previous rule. After that, it is necessary to add another row immediately after the block, with grey background, and labeled with ellipsis. Finally, in the first row of the block it is added a black horizontal line which expresses the limit of the expansion block.

The vertical composition of classes ($c \hat{} c$) is analogous to the vertical composition of blocks.

The horizontal composition of sheets ($s \mid s$) is analogous to the horizontal composition of blocks.

Finally, the horizontally expandable classes ($c^{\rightarrow}$) are embedded starting with the normal embedding of the class. The next step is to add an extra column immediately after the last column of the class, with grey background, and labeled with ellipsis. This expresses the expandability. The last step is to add a black vertical line which delimits the expansion.

Note the label constructors (Lab, Hor, and Ver) are not part of the embedding process, as they do not impact the visual model. Nevertheless, they are used by our algorithm to decide when two blocks are part of a greater one. This is also used to decide when blocks have the same color.

Given the embedding of the spreadsheet model in one worksheet, it is now possible to have one of its instances in a second worksheet, as we will discuss in the next sections. As we will also see, this setting has a couple of advantages: firstly, users may evolve the model having the data automatically coevolved. Secondly, having the model near the data helps to document the latter, since users can clearly identify the structure of the logic behind the spreadsheet. Figure 5a illustrates the complete embedding for the *ClassSheet* model of the running example, whilst Figure 5b shows one of its possible instances.

3.1 Model Creation

To create a model, several operations are available such as addition and deletion of columns and rows, cell editing, and addition and deletion of classes. To create, for example, the flights' part of the spreadsheet used so far, one can:

1. add a class for the flights, selecting the range `A1:G6` and choosing the green color for its background;
2. add a class for the planes, selecting the range `B1:F6`, choosing the cyan color for its background, and setting the class to expand horizontally;
3. add a class for the pilots, selecting the range `A3:G5`, choosing the yellow color for its background, and setting the class to expand vertically; and,

4. set the labels and formulas for the cells.

The addition of the relation class (range `B3:E4`) is not needed since it is automatically added when the environment detects superposing classes at the same level (**PlanesKey** and **PilotsKey** are within **Flights**, which leads to the automatic insertion of the relation class).

3.2 Instance Generation

From the flights' model described above, an *empty* spreadsheet instance can be generated. This is performed by copying the structure of the model to another worksheet. In this process labels are simply copied, and attributes are replaced in one of two ways:

1. If the attribute is simple (i.e., it is like $a = \varphi$), it is replaced by its default value. This default value is also used to set the instance cell type. After parsing and determining the type of the default value, the corresponding instance cells are set to the correct type. For this we use the spreadsheet system built-in type mechanism. Since this mechanism is quite flexible, it may happen that the instance has values of different types than the ones defined in the model. Nevertheless, our tool could be extended to guarantee such restrictions are enforced as we have done with other restrictions [17,18].
2. If the attribute is a formula, it is replaced by an instance of the formula. An instance of a formula is similar to the original one defined in the model, but the attribute references are replaced by references to cells where those attributes are instantiated.

Columns and rows with ellipses have no content, having instead buttons to perform operations of adding new instances of their respective classes. When pressed, new cells are created with the corresponding default values, and all the formulas are updated to accommodate the changes.

From the flights' model, we would obtain an initial instance that has the exact same structure as the spreadsheet shown in Figure 5b: the same labels and the same four buttons that are available to add new instances of the expandable classes. Compared to that spreadsheet, however, its initial version would hold no values except those from the default values defined in the model. Considering, for example, the **Pilots** table, this means that a single line would be present below the column headers, and that this line would have its **ID** and **Name** values set to the empty string and its **Phone** number set to 0.

3.3 Data Editing

The editing of the data is performed like with plain spreadsheets, i.e., the user just edits the cell content. The insertion of new data is different since editing assistance must be used through the buttons available.

For example, to insert a new flight for pilot `p11` in the **Flights** table, without models one would need to:

1. insert four new columns;
2. copy all the labels;
3. update all the necessary formulas in the last column; and,
4. insert the values for the new flight.

With a large spreadsheet, the step to update the formulas can be very error prone, and users may forget to update all of them. Using models, this process consists of only two steps:

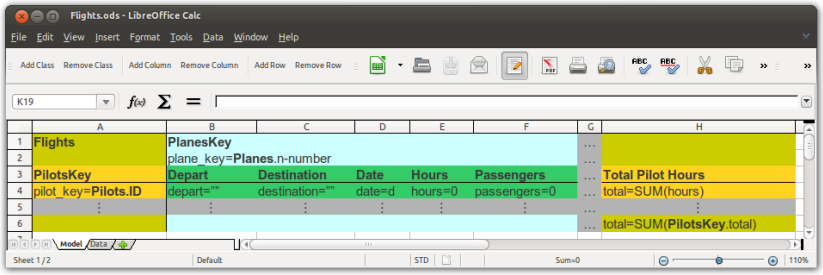
1. press the button with label "· · · " (in column J, Figure 5b); and,
2. insert the values for the new flight.

The model-driven environment automatically inserts four new columns, the labels for those columns, updates the formulas, and inserts default values in all the new input cells.

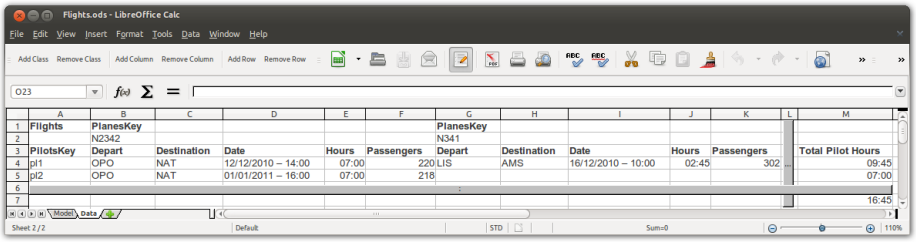
Note that, to keep the consistency between instance and model, all the cells in the instance that are not data entry cells are non-editable, that is, all the labels and formulas cannot be edited in the instance, only in the model. In Section 4 we will detail how to handle model evolutions.

4 Model-Driven Spreadsheet Evolution

The example we have been using manages pilots, planes and flights, but it misses a critical piece of information about flights: the number of passengers. In this case, additional columns need to be inserted in the block of each flight. Figure 6 shows an evolved spreadsheet with new columns (F and K) to store the number of passengers (Figure 6b), as well as the new model that it instantiates (Figure 6a). Note that a modification of the block that relates pilots and planes in the model (in this case, inserting a new column) captures modifications to all repetitions of the block throughout the instance.



(a) Evolved flights' model.



(b) Evolved flights' instance.

Figure 6: Evolved spreadsheet and the model that it instantiates.

In this section, we will demonstrate that modifications to spreadsheet models can be supported by an appropriate combinator language, and that these model modifications

can be propagated automatically to the spreadsheets that instantiate the models. In the case of the flights example, the model modification is captured by the following expression:

```
addPassengers =
  once (inside "PilotsKey_PlanesKey"
    (after "Hours" (insertCol "Passengers")))
```

The actual column insertion is done by the innermost $insertCol$ step. The $after$ and $inside$ combinators specify the location constraints of applying this step. The $once$ combinator traverses the spreadsheet model to search for a single location where these constraints are satisfied and the insertion can be performed.

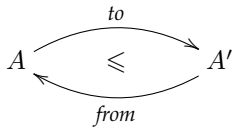
The application of $addPassengers$ to the initial model (Figure 5a) will yield:

1. the modified model (Figure 6a),
2. a spreadsheet migration function that can be applied to instances of the initial model (e.g. Figure 5b) to produce instances of the modified model (e.g. Figure 6b), and
3. an inverse spreadsheet migration function to backport instances of the modified model to instances of the initial model.

In the remainder of this section we will explain the machinery required for this type of coupled transformation of spreadsheet instances and models.

4.1 A Framework for Evolution of Spreadsheets in HASKELL

Data refinement theory provides an algebraic framework for calculating with data types and corresponding values [20–22]. 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). Figure 7 depicts the general scenario of a transformation in this framework.



A, A'	data type and transformed data type
to	witness function of type $A \to A'$ (injective)
$from$	witness function of type $A' \to A$ (surjective)

Figure 7: Coupled transformation of data type A into data type A'.

Each transformation is coupled with witness functions to and $from$, which are responsible for converting values of type A into type A' and back.

2LT is a framework written in HASKELL implementing this theory [23–26]. 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. To represent the witness functions *from* and *to* 2LT relies on the definition of a *Generalized Algebraic Data Type*[4] (GADT) [27, 28]. Each *from* and *to* function is represented by a value of the polymorphic type $PF\ a$, which can be defined using the following constructors:[5]

$$
\begin{array}{ll}
PF\ a ::= & \\
\quad | \ id : PF\ (a \to a) & \text{-- identity function} \\
\quad | \ \pi_1 : PF\ ((a, b) \to a) & \text{-- left projection of a pair} \\
\quad | \ \pi_2 : PF\ ((a, b) \to b) & \text{-- right projection of a pair} \\
\quad | \ pnt : a \to PF\ (One \to a) & \text{-- constant} \\
\quad | \ \cdot \triangle \cdot : PF\ (a \to b) \to PF\ (a \to c) \to PF\ (a \to (b, c)) & \text{-- split of functions} \\
\quad | \ \cdot \times \cdot : PF\ (a \to b) \to PF\ (c \to d) \to PF\ ((a, c) \to (b, d)) & \text{-- product of functions} \\
\quad | \ \cdot \circ \cdot : Type\ b \to PF\ (b \to c) \to PF\ (a \to b) \to PF\ (a \to c) & \text{-- composing of functions} \\
\quad | \ \cdot^{\star} : PF\ (a \to b) \to PF\ ([a] \to [b]) & \text{-- map of func.} \\
\quad | \ head : PF\ ([a] \to a) & \text{-- head of a list} \\
\quad | \ tail : PF\ ([a] \to [a]) & \text{-- tail of a list} \\
\quad | \ fhead : PF\ (VFormula \to RefCell) & \text{-- head of the arguments of a formula} \\
\quad | \ ftail : PF\ (VFormula \to FormulaV) & \text{-- tail of the arguments of a formula}
\end{array}
$$

This GADT represents the types of the functions used in the transformations. Each constructor has a name and its type. For instance, the pnt constructor receives a value of type a and returns a value of type $PF\ (One \to a)$, meaning it returns a representation of a function that has as argument a value of the singleton type and returns a value of type a, that is, it transforms a constant into a function which will always return that constant. Another example is π_1, which represents the type of the function that projects the first part of a pair. The comments should clarify which function each constructor represents.

Given these representations of types and functions, we can turn to the encoding of refinements. Each refinement is encoded as a two-level rewriting *rule*:

$$Rule = \forall\ a\ .\ Type\ a \to Maybe\ (View\ (Type\ a))$$

Although the refinement is from a type a to a type b, this cannot be directly encoded since the type b is only known when the transformation completes, so the type b is represented as a *view* of the type a. Since the the transformation may fail, it is wrapped in the $Maybe$ type. $Maybe$ encapsulates an optional value: a value of type $Maybe\ a$ either contains a value of type a ($Just\ a$), or it is empty ($Nothing$).

$$
\begin{array}{l}
View\ a ::= \\
\quad View : Rep\ a\ b \to Type\ b \to View\ (Type\ a)
\end{array}
$$

[4] "It allows to assign more precise types to data constructors by restricting the variables of the datatype in the constructors' result types."

[5] Although we use a notation similar to HASKELL, we try to keep it more abstract so it can more easily be used in other contexts.

A *view* expresses that a type a can be represented as a type b, denoted as $Rep\ a\ b$, if there are function representations $to : a \to b$ and $from : b \to a$ that allow data conversion between one and the other.

$$
\begin{array}{l}
Rep\ a\ b ::= \\
\quad Rep\ \{to = PF\ (a \to b), \{from = PF\ (b \to a)\}
\end{array}
$$

To better explain this system we will show a small example. The following code implements a rule, *listmap*, to transform a list into a map (represented by $\cdot \rightharpoonup \cdot$):

$$
\begin{array}{l}
listmap : Rule \\
listmap\ ([a]) = \\
\quad Just\ (\\
\quad\quad View\ (Rep\ \{to = seq2index, from = tolist\}) \\
\quad\quad\quad\quad (Int \rightharpoonup a) \\
\quad) \\
listmap\ \_ = mzero
\end{array}
$$

The witness functions have the following signature (for this example their code is not important):

$$
\begin{array}{ll}
tolist & : (Int \rightharpoonup a) \to [a] \\
seq2index & : [a] \to (Int \rightharpoonup a)
\end{array}
$$

This rule receives the type of a list of a, $[a]$, and returns a (maybe) view over the type map of integers to a, $Int \rightharpoonup a$. The witness functions are returned in the representation Rep. If some other argument than a list is received, then the rule fails returning $mzero$. All the rules contemplate this last case and so we will not show it in the definition of other rules.

Given this encoding of individual rewrite rules, a complete rewrite system can be constructed via the following constructors:

$$
\begin{array}{ll}
nop : Rule & \text{-- identity} \\
\triangleright : Rule \to Rule \to Rule & \text{-- sequential composition} \\
\oslash : Rule \to Rule \to Rule & \text{-- left-biased choice} \\
many : Rule \to Rule & \text{-- repetition} \\
once : Rule \to Rule & \text{-- arbitrary depth rule apply}
\end{array}
$$

Details on the implementation of these combinators can be found elsewhere [23].

4.1.1 *ClassSheets* and Spreadsheets in HASKELL

The 2LT was originally designed to work with algebraic data types. However, this representation is not expressive enough to represent *ClassSheet* specifications or their spreadsheet instances. To overcome this issue, we extended the 2LT representation so it could support *ClassSheet* models, by introducing the following GADT:

$Type\ a ::=$

-- previous shown constructors

...

-- plain spreadsheet value

$| Value : Value \to Type\ Value$

-- references

$| Ref : Type\ b \to PF\ (a \to RefCell)$
$\to PF\ (a \to b) \to Type\ a \to Type\ a$

-- reference cell

$| RefCell : Type\ RefCell$

-- formulas

$| Formula : VFormula \to Type\ VFormula$

-- block label

$| LabelB : String \to Type\ LabelB$

-- attributes

$| \cdot = \cdot : Type\ a \to Type\ b \to Type\ (a, b)$

-- block horizontal composition

$| \cdot \shortmid \cdot : Type\ a \to Type\ b \to Type\ (a, b)$

-- block vertical composition

$| \cdot \hat{} \cdot : Type\ a \to Type\ b \to Type\ (a, b)$

-- empty block

$| EmptyB : Type\ EmptyB$

-- horizontal class label

$| \underline{\cdot} : String \to Type\ HorH$

-- vertical class label

$|| \cdot : String \to Type\ VerV$

-- square class label

$|| \underline{\cdot} : String \to Type\ Square$

-- relation class

$| LabRel : String \to Type\ LabS$

-- labeled class

$| \cdot : \cdot : Type\ a \to Type\ b \to Type\ (a, b)$

-- labeled expandable class

$| \cdot : (\cdot)^{\downarrow} : Type\ a \to Type\ b \to Type\ (a, [b])$

-- class vertical composition

$| \cdot \hat{} \cdot : Type\ a \to Type\ b \to Type\ (a, b)$

-- sheet class

$| SheetC : Type\ a \to Type\ (SheetC\ a)$

-- sheet expandable class

$| \cdot^{\rightarrow} : Type\ a \to Type\ [a]$

-- sheet horizontal composition

$| \cdot \shortmid \cdot : Type\ a \to Type\ b \to Type\ (a, b)$

-- empty sheet

$| EmptyS : Type\ EmptyS$

The constructors of this data type represent each of the elements of the textual syntax of *ClassSheets* presented in Figure 1. Indeed we try to use the same notation to keep our representation as similar to the original language as possible. The comments should help match the *ClassSheet* language and the constructors. The values of type $Type\ a$ are representations of type a. For example, if t is of type $Type\ Value$, then t represents the type $Value$. The following types are needed to construct values of type $Type\ a$:

```
EmptyBlock                          -- empty block
EmptySheet                          -- empty sheet
LabelB = String                     -- label
RefCell = RefCell1                  -- referenced cell
LabS = String                       -- square label
HorH = String                       -- horizontal label
VerV = String                       -- vertical label
SheetC a ::=                        -- sheet class
  SheetCC a
SheetCE a ::=                       -- expandable sheet class
  SheetCEC a
Value ::=                           -- value
    VInt Int
  | VString String
  | VBool Bool
  | VDouble Double
VFormula ::=                        -- formula
    FValue Value
  | FRef
  | FFormula String [VFormula]
```

Once more, the comments should clarify what each type represents. To explain this representation we will use as an example a small table representing the costs of maintenance of planes. We do not use the running example as it would be very complex to explain and understand. For this reduced model only four columns were defined: *plane model*, *quantity*, *cost per unit* and *total cost* (product of *quantity* by *cost per unit*). The HASKELL representation of such a model is presented in the next code listing.

```
costs =
| Cost : Model ¦ Quantity ¦ Price ¦ Total^
| Cost : (model = "" ¦ quantity = 0 ¦ price = 0 ¦ total = FFormula "×" [FRef, FRef])↓
```

This *ClassSheet* specifies a class called $Cost$ composed by two parts vertically composed as indicated by the ^ operator. The first part is specified in the first row and defines the labels for four columns: $Model$, $Quantity$, $Price$ and $Total$. The second row models the rest of the class containing the definition of the four columns. The first column has default value the empty string (`""`), the two following columns have as default value 0, and the last one is defined by a formula (explained latter on). Note that this

part is vertically expandable. Figure 8 represents a spreadsheet instance of this model.

	A	B	C	D
1	Model	Quantity	Price	Total
2	B747	2	1500	=B2*C2
3	B777	5	2000	=B3*C3

Figure 8: Spreadsheet instance of the maintenance costs *ClassSheet*.

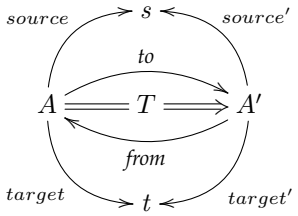
Note that in the definition of $Type\ a$ the constructors combining parts of the spreadsheet (e.g. sheets) return a pair. Thus, a spreadsheet instance is written as nested pairs of values. The spreadsheet illustrated in Figure 8 is encoded in HASKELL as follows:

$$
\begin{array}{l}
((Model, (Quantity, (Price, Total))),\\
[(\texttt{"B747"}, (2, (1500, FFormula\ \texttt{"}\times\texttt{"}\ [FRef, FRef]))),\\
\ (\texttt{"B777"}, (5, (2000, FFormula\ \texttt{"}\times\texttt{"}\ [FRef, FRef])))])
\end{array}
$$

The HASKELL type checker statically ensures that the pairs are well formed and are constructed in the correct order.

4.1.2 Specifying References

Having defined a GADT to represent *ClassSheet* models, we now need a mechanism to define spreadsheet references. The safest way to accomplish this is making references strongly typed. Figure 9 depicts the scenario of a transformation with references. A reference from a cell s to a cell t is defined using a pair of projections, $source$ and $target$. These projections are statically-typed functions traversing the data type A to identify the cell defining the reference (s), and the cell to which the reference is pointing to (t). In this approach, not only the references are statically typed, but also always guaranteed to exist, that is, it is not possible to create a reference from/to a cell that does not exist.



$source$	Projection over type A identifying the reference
$target$	Projection over type A identifying the referenced cell

$$source' = source \circ from$$
$$target' = target \circ from$$

Figure 9: Coupled transformation of data type A into data type A' with references.

The projections defining the reference and the referenced type, in the transformed type A', are obtained by post-composing the projections with the witness function $from$.

When $source'$ and $target'$ are normalized they work on A' directly rather than via A. The formula specification, as previously shown, is specified directly in the GADT. However, the references are defined separately by defining projections over the data type. This is required to allow any reference to access any part of the GADT.

Using the spreadsheet illustrated in Figure 8, an instance of a reference from the formula *total* to *price* is defined as follows (remember that the second argument of Ref is the source (reference cell) and that the third is the target (referenced cell)):

$$\begin{aligned} &costWithReferences = \\ &\quad Ref\ Int\ (fhead \circ head \circ (\pi_2 \circ \pi_2 \circ \pi_2)^\star \circ \pi_2)\ (head \circ (\pi_1 \circ \pi_2 \circ \pi_2)^\star \circ \pi_2)\ cost \end{aligned}$$

The $source$ function refers to the first $FRef$ in the HASKELL encoding shown after Figure 8. The $target$ projection defines the cell it is pointing to, that is, it defines a reference to the the value 1500 in column $Price$.

To help understand this example, we explain how $source$ is constructed. Since the use of GADTs requires the definition of models combining elements in a pairwise fashion, π_2 is used to get the second element of the model (a pair), that is, the list of planes and their cost maintenance. Then, we apply $(\pi_2 \circ \pi_2 \circ \pi_2)^\star$ which will return a list with all the formulas. Finally $head$ will return the first formula (the one in cell D2) from which $fhead$ gets the first reference in a list of references, that is, the reference B2 that appears in cell D2.

Note that our reference type has enough information about the cells and thus we do not need value-level functions, that is, we do not need to specify the projection functions themselves, just their types. In the cases we reference a list of values, for example, constructed by the class expandable operator, we need to be specific about the element within the list we are referencing. For these cases, we use the type-level constructors $head$ (first element of a list) and $tail$ (all but first) to get the intended value in the list.

4.2 Evolution of Spreadsheets

In this section we define rules to perform spreadsheet evolution. These rules can be divided in three main 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).

4.2.1 Combinators

The semantic and the layout rules are defined to work on a specific part of the model. The combinators defined next are then used to apply those rules in the desired places.

Pull up all references: To avoid having references in different levels of the models, all the rules pull all references to the topmost level of the model. This allows to create simpler rules since the positions of all references are known and do not need to be changed when the model is altered. To pull a reference in a particular place we use the following rule (we just show its first case):

$$\begin{aligned} &pullUpRef : Rule \\ &pullUpRef\ ((Ref\ tb\ fRef\ tRef\ ta) \mid b2) = \textbf{return} \\ &\quad View\ idrep\ (Ref\ tb\ (fRef \circ \pi_1)\ (tRef \circ \pi_1)\ (ta \mid b2)) \end{aligned}$$

The representation $idrep$ has the id function in both directions. If part of the model (in this case the left part of a horizontal composition) of a given type has a reference, it is pulled to the top level. This is achieved by composing the existing projections with the necessary functions, in this case π_1. This rule has two cases (left and right hand side) for each binary constructor (e.g. horizontal/vertical composition).

To pull up all the references in all levels of a model we use the rule

$$pullUpAllRefs = many\ (once\ pullUpRef)$$

The $once$ operator applies the $pullUpRef$ rule somewhere in the type and the $many$ ensures that this is applied everywhere in the whole model.

Apply after and similars: The combinator $after$ finds the correct place to apply the argument rule (second argument) by comparing the given string (first argument) with the existing labels in the model. When it finds the intended place, it applies the rule to it. This works because our rules always do their task on the right-hand side of a type.

$$\begin{array}{l} after : String \to Rule \to Rule \\ after\ label\ rule\ (label' \mid b) = \\ \quad \textbf{if}\ label \equiv label' \\ \quad\quad View\ s\ l' \leftarrow rule\ label' \\ \quad\quad \textbf{return}\ View\ (Rep\ \{to = (to\ s) \times id, \\ \quad\quad\quad\quad\quad\quad\quad\quad from = (from\ s) \times id\}) \\ \quad\quad\quad\quad\quad\quad (l' \mid b)) \end{array}$$

After comparing the argument label ($label$) with the label from the model ($label'$), if they are equal, it applies the rule and then returns the updated model type. Note that this code represents only part of the complete definition of the function. The remaining cases, e.g. $\cdot\hat{}\cdot$, are not shown since they are quite similar to the one presented.

Other combinators were also developed, namely, $before$, $below$, $above$, $inside$ and at. Their implementations are not shown since they are similar to the $after$ combinator.

4.2.2 Semantic Rules

Given the support to apply rules in any place of the model given by the previous definitions, we now present rules that change the semantics of the model, that is, that change the meaning and the model itself, e.g., adding columns.

Insert a block: The first rule we present is one of the most fundamental: the insertion of a new block into a spreadsheet. It is formally defined as follows:

$$Block \underset{\pi_1}{\overset{id\triangle(pnt\ a)}{\leqslant}} Block \mid Block$$

This diagram means that a horizontal composition of two blocks refines a block when witnessed by two functions, to and $from$. The to function, $id\triangle(pnt\ a)$, is a split: it injects the existing block in the first part of the result without modifications (id) and

injects the given block instance a into the second part of the result. The $from$ function is π_1 since it is the one that allows the recovery of the existent block. The HASKELL version of the rule is presented next.

$$
\begin{array}{l}
insertBlock : Type\ a \to a \to Rule \\
insertBlock\ ta\ a\ t = \\
\quad \textbf{if}\ (isBlock\ ta) \wedge (isBlock\ t) \\
\quad\quad rep \leftarrow Rep\ \{ to = id\triangle(pnt\ a), from = \pi_1 \} \\
\quad\quad View\ s\ t' \leftarrow pullUpAllRefs\ (t \vartriangleright ta) \\
\quad\quad \textbf{return}\ View\ (comprep\ rep\ s)\ t'
\end{array}
$$

The function $comprep$ composes two representations. This rule receives the type of the new block ta, its default instance a, and returns a $Rule$. The returned rule is itself a function that receives the block to modify t, and returns a view of the new type. The first step is to verify if the given types are blocks using the function $isBlock$. The second step is to create the representation rep with the witness functions given in the above diagram. Then the references are pulled up in result type $t \vartriangleright ta$. This returns a new representation s and a new type t' (in fact, the type is the same $t' = t \vartriangleright ta$). The result view has as representation the composition of the two previous representations, rep and s, and the corresponding type t'.

Rules to insert classes and sheets were also defined, but since these rules are similar to the rule to insert blocks, we omit them.

Insert a column: To insert a column in a spreadsheet, that is, a cell with a label $labl$ and the cell below with a default value $form$ and vertically expandable, we first need to create a new class representing it:

$$| labl : labl\hat{}((labl = form)^{\downarrow})$$

The label is used to create the default value $(labl, [\,])$. Note that since we want to create an expandable class, the second part of the pair must be a list. The final step is to apply $insertSheet$:

$$
\begin{array}{l}
insertCol : String \to VFormula \to Rule \\
insertCol\ labl\ form\ sh = \\
\quad \textbf{if}\ isSheet\ sh \\
\quad\quad clas \leftarrow (| labl : labl\hat{}((labl = form)^{\downarrow})) \\
\quad\quad \textbf{return}\ ((insertSheet\ clas\ (labl, [\,])) \rhd pullUpAllRefs)\ sh
\end{array}
$$

Note the use of the rule $pullUpAllRefs$ as explained before. The case shown in the above definition is for a formula as default value and it is similar to the value case. The case with a reference is more interesting and is shown next:

$$
\begin{array}{l}
insertCol\ labl\ FRef\ sh = \textbf{if}\ isSheet\ sh \\
\quad clas \leftarrow (| labl : Ref\ \bot\ \bot\ \bot\ (labl\hat{}((labl = RefCell)^{\downarrow}))) \\
\quad \textbf{return}\ ((insertSheet\ clas\ (labl, [\,])) \rhd pullUpAllRefs)\ sh
\end{array}
$$

Recall that our references are always local, that is, they can only exist with the type they are associated with. So, it is not possible to insert a column that references a part of

the existing spreadsheet. To overcome this, we first create the reference with undefined functions and auxiliary type ($\perp$). We then set these values to the intended ones.

```
setFormula : Type b → PF (a → RefCell) → PF (a → b) → Rule
setFormula tb fRef tRef (Ref _ _ _ t) =
  return View idrep (Ref tb fRef tRef t)
```

This rule receives the auxiliary type ($Type\ b$), the two functions representing the reference projections and adds them to the type. A complete rule to insert a column with a reference is defined as follows:

```
insertFormula =
  (once (insertCol label FRef)) ▷ (setFormula auxType fromRef toRef)
```

Following the original idea described previously in this section, we want to introduce a new column with the number of passengers in a flight. In this case, we want to insert a column in an existing block and thus our previous rule will not work. For these cases we write a new rule:

```
insertColIn : String → VFormula → Rule
insertColIn labl (FValue v) b =
  if isBlock b
    block ← labl^(labl = v)
    return ((insertBlock block (labl, v)) ▷ pullUpAllRefs) b
```

This rule is similar to the previous one but it creates a block (not a class) and also inserts it after a block. The reasoning is analogous to the one in $insertCol$.

To add the column `"Passengers"` we can use the rule $insertColIn$, but applying it directly to our running example will fail since it expects a block and we have a spreadsheet. We can use the combinator $once$ to achieve the desired result. This combinator tries to apply a given rule somewhere in a type, stopping after it succeeds once. Although this combinator already existed in the 2LT framework, we extended it to work for spreadsheet models/types.

Make it expandable: It is possible to turn a regular block within a class into an expandable block. For this, we created the rule $expandBlock$:

```
expandBlock : String → Rule
expandBlock lb (label : clas) =
  if lb ≡ label
    rep ← Rep {to = id × tolist, from = id × head}
    return View rep (label : (clas)↓)
```

It receives the label of the class to make expandable and updates the class to allow repetition. The result type constructor is $\cdot : (\cdot)^{\downarrow}$; the to function wraps the existing block into a list, $tolist$; and the $from$ function takes the head of it, $head$. We developed a similar rule to make a class expandable. This corresponds to a promotion of a class c to $c^{\rightarrow}$. We do not show its implementation here since it is quite similar to the one just shown.

Split: It is quite common to move a column in a spreadsheet from on place to another. The rule *split* copies a column to another place and substitutes the original column values by references to the new column (similar to creating a pointer). The rule to move part of the spreadsheet is presented in Section 4.2.3. The first step of *split* is to get the column that we want to copy:

$$
\begin{aligned}
&getColumn : String \to Rule \\
&getColumn\ lb\ (label \hat{}\ b) = \\
&\quad \textbf{if}\ lb \equiv label \\
&\qquad \textbf{return}\ View\ idrep\ (label \hat{}\ b)
\end{aligned}
$$

If the corresponding label is found, the vertical composition is returned. Note that as in other rules, this one is intended to be applied using the combinator *once*. As we said, we aim to write local rules that can be used at any level using the developed combinators.

In a second step the rule creates a new a class (*nclass*) containing the retrieved block:

$$
\begin{aligned}
&View\ rep\ c' \leftarrow getColumn\ lb\ c \\
&nclass \leftarrow (|\ lb : (c')^{\downarrow})
\end{aligned}
$$

The last step is to transform the original column that was copied into references to the new column. The rule $makeReferences : String \to Rule$ receives the label of the column that was copied (the same as the new column) and creates the references. We do not show the rest of the implementation because it is quite complex and will not help in the understanding of the paper.

4.2.3 Layout Rules

We will now describe rules focused on the layout of spreadsheets, that is, rules that do not add/remove information to/from the model, but only rearrange it.

Change orientation The rule *toVertical* changes the orientation of a block from horizontal to vertical.

$$
\begin{aligned}
&toVertical : Rule \\
&toVertical\ (a \mid b) = \textbf{return}\ View\ idrep\ (a \hat{}\ b)
\end{aligned}
$$

Note that since our value-level representation of these compositions are pairs, the *to* and the *from* functions are simply the identity function. The needed information is kept in the type-level with the different constructors. A rule to do the inverse was also designed but since it is quite similar to this one, we do not show it here.

Normalize blocks When applying some transformations, the resulting types may not have the correct shape. A common example is to have as result the following type:

$$
\begin{aligned}
&A \mid B \hat{}\ C \mid D \hat{} \\
&E \mid F
\end{aligned}
$$

However, given the rules in [7] to ensure the correctness of *ClassSheets*, the correct result is the following:

$A \mid B \mid D\hat{}$
$E \mid C \mid F$

The rule *normalize* tries to match these cases and correct them. The types are the ones presented above and the witness functions are combinations of π_1 and π_2.

```
normalize : Rule
normalize (a | b^c | d^e | f) =
  to ← id × π1 × id ∘ π1△π1 ∘ π2△π2 ∘ π1 ∘ π2 × π2
  from ← π1 ∘ π1△π1 ∘ π2 × π1 ∘ π2△π2 ∘ π2 ∘ π1△id × π2 ∘ π2
  return View (Rep {to = to, from = from}) (a | b | d^e | c | f)
```

Although the migration functions seem complex, they just rearrange the order of the pairs so they have the correct arrangement.

Shift: It is quite common to move parts of the spreadsheet across it. We designed a rule to shift parts of the spreadsheet in the four possible directions. We show here part of the *shiftRight* rule, which, as suggested by its name, shifts a piece of the spreadsheet to the right. In this case, a block is moved and an empty block is left in its place.

```
shiftRight : Type a → Rule
shiftRight ta b =
  if isBlock b
    Eq ← teq ta b
    rep ← Rep {to = pnt (⊥ :: EmptyBlock)△id,
               from = π2}
    return View rep (EmptyBlock | b)
```

The function *teq* verifies if two types are equal. This rule receives a type and a block, but we can easily write a wrapper function to receive a label in the same style of *insertCol*.

Another interesting case of this rule occurs when the user tries to move a block (or a sheet) that has a reference.

```
shiftRight ta (Ref tb frRf toRf b) =
  if isBlock b
    Eq ← teq ta b1
    r ← Rep {to = pnt (⊥ :: EmptyBlock)△id,
             from = π2}
    return View r (Ref tb (frRf ∘ π2) (toRf ∘ π2) (EmptyBlock | b)
```

As we can see in the above code, the existing reference projections must be composed with the selector π_2 to allow to retrieve the existing block *b1*. Only after this it is possible to apply the defined selection reference functions.

Move blocks: A more complex task is to move a part of the spreadsheet to another place. We present next a rule to move a block.

```
moveBlock : String → Rule
```

```
moveBlock str c =
  View s c' ← getBlock str c
  nsh ← (| str : c')
  View r sh ← once (removeRedundant str) (c ⫞ nsh)
  return View (comprep s r) sh
```

After getting the intended block and creating a new class with it, we need to remove the old block using *removeRedundant*.

```
removeRedundant : String → Rule
removeRedundant s (s')
  if s ≡ s'
    rep ← Rep { to = pnt (⊥ :: EmptyBlock),
                from = pnt s' }
    return View rep EmptyBlock
```

This rule will remove the block with the given label leaving an empty block in its place.

5 The MDSheet Framework

The embedding and evolution techniques previously presented have been implemented as an add-on to a widely used spreadsheet system, the OpenOffice/LibreOffice system. The add-on provides a model-driven spreadsheet development environment, named MDSheet, where a (model-driven) spreadsheet consists of two types of worksheets: `Sheet 0`, containing the embedded *ClassSheet* model, and `Sheet 1`, containing the spreadsheet data that conforms to the model. Users can interact with both the *ClassSheet* model and the spreadsheet data. Our techniques guarantee the synchronization of the two representations.

In such a model-driven environment, users can evolve the model by using standard editing/updating techniques as provided by spreadsheets systems. Our add-on/environment also provides predefined buttons that implement the usual *ClassSheets* evolution steps. Each button implements an evolution rule, as described in Section 4. For each button, we defined a BASIC script that interprets the desired functionality, and sends the contents of the spreadsheet (both the model and the data) to our HASKELL-based co-evolution framework. This HASKELL framework implements the co-evolution of the spreadsheet models and data presented in Section 4.

MDSheet also allows the development of *ClassSheet* models from scratch by using the provided buttons or by traditional editing. In this case, a first instance/spreadsheet is generated from the model which includes some business logic rules that assist users in the safe and correct introduction/editing of data. For example, in the spreadsheet presented in Figure 5b, if the user presses the button in column `J`, four new columns will automatically be inserted so the user can add more flights. This system will also automatically update all formulas in the spreadsheet.

The global architecture of the model-driven spreadsheet development we constructed is presented in Figure 10.

Tool and demonstration video availability: The MDSheet tool and a video with a demonstration of its capabilities are available at

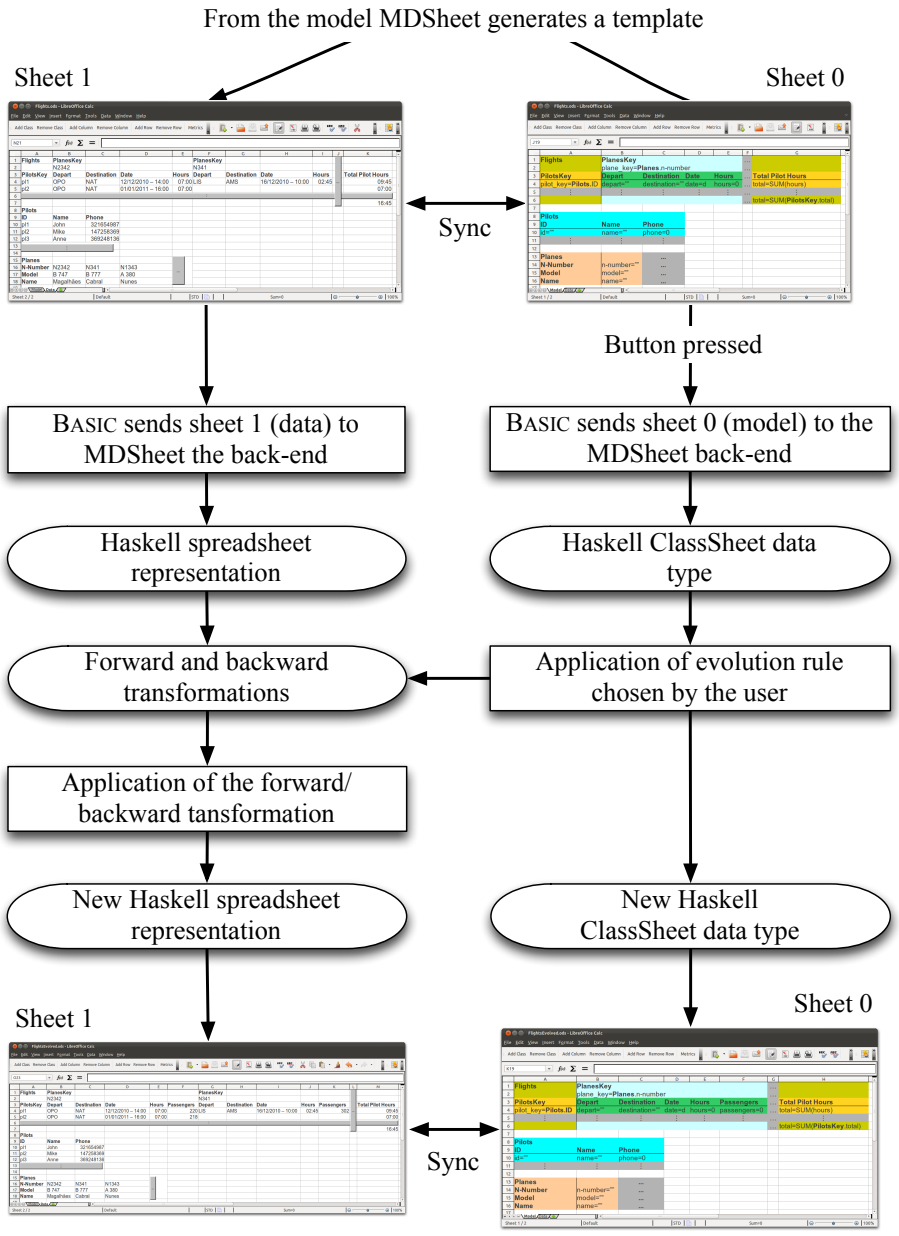


Figure 10: Model-driven spreadsheet development environment.

`http://ssaapp.di.uminho.pt`.

In the next section we present in detail the empirical study we have organized and conducted to assess model-driven spreadsheets running through MDSheet.

6 Empirical Evaluation

In the context of software engineering research, empirical validation is widely recognized as essential in order to assess the validity of newly proposed techniques or methodologies [29]. While our previous work on the construction of a model-driven spreadsheet development environment has received good feedback from the research community, the fact is that its assessment in a realistic and practical context was still lacking. In this line, we have designed an empirical study that we describe in this section and whose results we also analyze in detail here.

The experiment that we envisioned is motivated by the need to understand the differences in individual performance that users achieve under MDSheet (using what we call *model-driven* spreadsheets) against traditional spreadsheet systems (from now on termed *plain*). The perspective of the experiment is from the point of view of a researcher who would like to know whether there is a systematic difference in the users performance.

In this section we detail the different stages that we underwent in preparing and designing our study (Section 6.1), in running it (Section 6.2), and in analyzing (Section 6.3) and interpreting (Section 6.4) the results, which are discussed afterwards (Section 6.5). Finally, we can summarize the scope of this study as suggested in [29] as follows:

> Analyze *the spreadsheet development process*
> for the purpose of *evaluation*
> with respect to its *effectiveness and efficiency*
> from the point of view of the *researcher*
> in the context of *the usage of two different spreadsheets by Master students*.

6.1 Design

The goal of our study is to analyze several aspects of spreadsheet development, and to evaluate the implications of using a model-driven approach against the more commonly used approach of designing and introducing spreadsheet data from scratch and immediately within spreadsheets themselves.

The study that we conducted was designed for a controlled environment mostly because our tool was never tested in production. In order to achieve this controlled environment, we decided to perform the study in an off-line setting (in an academic environment and not in industry), and with university students attending a Master's program. Furthermore, our study analyzes the specific use of *ClassSheet*-based models, and does not consider generic model-driven spreadsheet development. Finally, in our study, participants were asked to solve realistic problems, in situations that were closely adapted from real-world situations.

6.1.1 Hypotheses

MDSheet uses *ClassSheet*-based models to specify spreadsheets, hence it benefits from *ClassSheet* advantages such as: (i) users are freed from the risks associated with editing formulas directly, and (ii) users do not have to manually identify parts of the spreadsheet that are repeatable (class expansions). In theory, (i) reduces the number of errors and (ii) improves spreadsheet development performance. However, this needs to be tested. Thus, we can informally state two hypotheses:

1. In order to perform a given set of tasks, users spend less time when using model-driven spreadsheets instead of plain ones.
2. Spreadsheets developed in the model-driven environment contain less errors than plain ones.

Formally, two hypotheses are being tested: H_T for the time that is needed to perform a given set of tasks, and H_E for the error rate found in different types of spreadsheets. They are respectively formulated as follows:

1. *Null hypothesis*, H_{T_0}: The time to perform a given set of tasks using MDSheet is not less than that taken with plain spreadsheets. $H_{T_0} : \mu_d \leqslant 0$, where μ_d is the expected mean of the time differences.

   Alternative hypothesis, $H_{T_1} : \mu_d > 0$, i.e., the time to perform a given set of tasks using MDSheet is less than with plain spreadsheets.

   Measures needed: time taken to perform the tasks.

2. *Null hypothesis*, H_{E_0}: The error rate in spreadsheets when using MDSheet is not smaller than with plain spreadsheets. $H_{E_0} : \mu_d \leqslant 0$, where μ_d is the expected mean of the differences of the error rates.

   Alternative hypothesis, $H_{E_1} : \mu_d > 0$, i.e., the error rate when using MDSheet is smaller than with plain spreadsheets.

   Measures needed: error rate for each spreadsheet.

6.1.2 Variables

The independent variables are: for H_T the *time to perform the tasks*, and for H_E the *error rate*.

6.1.3 Subjects and Objects

The subjects for this study were first year Master students undergoing a course at Universidade do Minho. Out of a total number of thirty-five students that were invited, twenty-five actually accepted the invitation and participated in our study. More details about the subjects participating in the study are presented in Section 6.3.

The objects for the study consisted in three different kinds of spreadsheets that are described later in this paper, in Section 6.1.5. One spreadsheet was used to support an in-study tutorial that was given to participants before they were actually asked to perform a series of tasks on the model-driven versions of the remaining two spreadsheets. This design choice attempts to minimize the threats to construct validity, namely the mono-operation bias (please refer to Section 6.4.1 for more details on threats to validity).

6.1.4 Design

In our study, we followed a standard design with one factor and two treatments, as presented in [29]. The factor is *the development method*, that is, spreadsheet development without using MDSheet. The treatments are *plain* and *model-driven*. The dependent variables are measurable in a ratio scale, and hence a parametric test is suitable.

Moreover, blocking is provided in the sense that each hypothesis is tested independently for each object. This enables to reduce the impact of the differences between the two spreadsheets.

6.1.5 Instrumentation

As we have been describing, our study was supported by three distinct kinds of spreadsheets. For the spreadsheet that was used in the tutorial, we have only constructed its model-driven version, but for the two remaining spreadsheets we have used both their model-driven and plain versions.

The spreadsheet that was used in the tutorial was designed to record (simplified) information on the flights of an airline company, namely its planes, their crew and the meals available on-board. As for the two remaining kinds of spreadsheets, they were selected based on their practical interest: one of them, from now on termed *budget*, is an adapted version of the *Personal budget worksheet* that is available from Microsoft Office's webpage[6] (this spreadsheet has been downloaded over 4 million times); the other, *payments* is an adapted version of a spreadsheet that is being used to register the entire information regarding the payments that occur in the municipal company *Agere*[7] that is responsible for the water supply in the city of Braga, Portugal. In order to be usable, we have reduced the size of both spreadsheets, without changing their complexity.

The given budget spreadsheet had 66 rows and 80 columns with 3306 filled cells. It contained the information for 12 months of 6 consecutive years, organized in 11 different categories of income/expenses subdivided in income/expense items. Each year and category had a subtotal, and there was a grand total for the years and another for the categories, all of them being formulas.

The given payments spreadsheet had 33 rows and 55 columns with 1645 filled cells. It contained the information for 3 years, subdivided in 9 payment forms with 6 kinds of totals, and 1 month (January) with 31 days. At the end of each month/year there was also a grand total.

Guidelines were also provided to participants: they consisted of the list of tasks to be performed. For both spreadsheets, three tasks were given (non-essential data is omitted, being replaced by "[...]"):

Budget

i) "Add to the budget two new categories of expenses, [...], with the following expenses [...]."

ii) "Add a new year, [...], to the budget keeping the structure of the spreadsheet, and insert the following data: [...]."

iii) "Delete the information from categories [...]."

Payments

i) "Add a new month, [...], maintaining the structure of the spreadsheet and add the following data: [...]."

ii) "Add a new year, [...], keeping the structure of the spreadsheet and insert the following data: [...]."

iii) "Change the spreadsheet in order to remove the information related to kind of payment [...], removing the corresponding column."

[6] http://office.microsoft.com/en-us/templates/personal-budget-worksheet-TC006206279.aspx

[7] http://www.agere.pt/

Serveral versions of the lists of tasks were prepared, where each version had a specific task order, thus each participant performed the tasks sequencially in a random order. The data to be inserted did not contain more than six values per task. The participants had to update the formulas to include new references to cells when needed. The full description of the six tasks, that include the data to be added to the spreadsheets, is available at the tool webpage presented earlier.

To evaluate the participants' work, each task was sub-divided in small parts equivalent to each "atomic" operation that they should perform. For the budget spreadsheet, we have the following sub-division, with a total of twenty-three items:

- ten items for i), corresponding to inserting the lines for the data, inserting the values, and updating the formulas;
- nine items for ii), corresponding to inserting the new cells, inserting the data, inserting the formulas, and updating the formulas for the totals;
- four items for iii), corresponding to removing the categories, and updating the formulas.

For the payments spreadsheet, we have the following sub-division, with a total of ten items:

- four items for i), corresponding to inserting new cells for the data, inserting the data, and updating the formulas;
- four items for ii), corresponding to inserting new cells for the data, inserting the data, and updating the formulas;
- two items for iii), corresponding to deleting the columns and updating the formulas.

In order to understand the background of the subjects and the difficulties they experienced when participating in the study, two questionnaires were prepared: one answered before the study itself (pre-questionnaire) and another after (post-questionnaire).

The data collected consists of the modified spreadsheets by the participants, and some information about the performance of our model-driven environment. For that, the MDSheet add-on was modified in order to provide a log of the user actions when working with the model-driven spreadsheet. This log contains the action performed (e.g., "add instance" and "remove class"), and how much time the system took for each action.

6.1.6 Data Collection Procedure

Several steps were planned to run the study, with two distinct options in the order they should be performed. One of them is:

1. Filling the pre-questionnaire.
2. Performing the sets of tasks on the two plain spreadsheets, with a time limit of fifteen minutes.
3. Attending the tutorial on MDSheet.
4. Performing the sets of tasks on the two model-driven spreadsheets, with a time limit of fifteen minutes.
5. Filling the post-questionnaire.
6. Collecting all spreadsheets, questionnaires and logs.

The other option is: (1), (3), (4), (2), (5), and (6). This option consists of first performing the operations with the model-driven environment and then the operations on plain spreadsheets, as opposed to the first option in which the operations on plain spreadsheets is first. This design choice attempts to minimize the learning effects between treatments and so the validity threats.

In steps (3) and (6), our team was expected to have a direct participation, giving the tutorial in step (3) and retrieving, in step (6), all the artifacts used or created by the subjects.

All the subjects of our study were expected to perform the two sets of tasks on the respective spreadsheets. The goal of the study is not to compare one spreadsheet against another, but instead to compare two methods to develop spreadsheets. Furthermore, using two spreadsheets permits to reduce the mono-operation bias (see Section 6.4.1).

6.1.7 Analysis Procedure and Evaluation of Validity

The analysis of the collected data is achieved performing paired tests where the performance of each subject on the plain version of the spreadsheet is tested against the model-driven version. For this, the following tests are available: paired t-test, Wilcoxon sign rank test, and the dependent-samples sign-test.

To ensure the validity of the data collected, several kinds of support were planned: constant availability to clarify any doubt, tutorial to teach the model-driven development process, and slightly supervise the work done by the subjects in a way that do not interfere with their work. This last point consists of navigating through the room and see which subjects look like they are having problems and try to help them if it is about something that does not influence the results of the study.

6.2 Execution

The study was performed in two classrooms with twenty-five university students, eleven in one room and fourteen in the other one. The participants were randomly assigned to each room. All performed the study at the same time, but with different execution orders depending on the room that they were in.

The participants first started filling the pre-questionnaire, with generic information about themselves (gender, age range, and undergraduate major). They also answered some questions so we could assess their previous experience with spreadsheets.

While they filled the questionnaire, we checked if their environment was correctly set. This environment consisted in a virtual machine with Lubuntu 11.10 as the installed operating system, where we pre-installed LibreOffice Calc, and our add-on – MDSheet.

The list of tasks to be performed was then distributed amongst the participants, and they had fifteen minutes to perform all the tasks on the spreadsheets, but without the assistance of our framework.

Before telling the participants to perform the tasks on the spreadsheets with the models, they attended the tutorial that we prepared on how to use the MDSheet framework. During the tutorial, we answered all the questions that the participants had, making sure that they could use the framework.

The subjects in one room first performed the tasks with plain spreadsheets, then attended the tutorial and then performed the tasks on model-driven spreadsheets. The subjects in the other room first attended the tutorial, then performed the tasks on model-driven spreadsheets, and finally perfomed the tasks on plain spreadsheets.

At the end, we gave the post-questionnaire to the participants to evaluate the confidence that they had on their performance during the study. Finally, we collected the modified spreadsheet files, so that we could analyze them later on.

6.3 Analysis

6.3.1 Descriptive Statistics

Subjects: Basic information about the subjects was gathered, namely their gender, age, studies' background, and familiarity with spreadsheets. From the twenty-five subjects, twenty-one are male and four are female. Most of them are aged between twenty and twenty-eight, with two subject being over thirty-one. The subjects come from different areas, most of them having a background in *informatics engineering* or *computer science*, but others come from *information technology and communication*, *IT for health*, or *IT management*. Two of the subjects never worked with spreadsheets previously and the levels of experience vary from having used at least once to an heavy usage.

Time spent: As expected, differences were found in the time that subjects used to perform the tasks. The minimum times recorded on each spreadsheet were by participants using the model-driven environment, with average times being lesser for the model-driven approach.

Figure 11 shows the time each participant took to achieve the given tasks, both with and without the model-driven environment, for the budget spreadsheet. Only the results for the subjects that performed all the tasks are displayed to allow for an easier comparison.

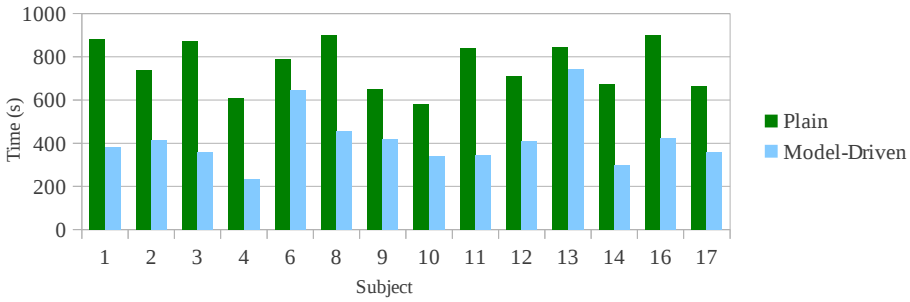


Figure 11: Time used to perform the tasks on the budget spreadsheet.

Similar results were obtained for the payments spreadsheet, as shown in Figure 12.

Error rates: To evaluate the correctness of the spreadsheets produced during the study, error rates are used. Each of the six tasks requires a set of spreadsheet operations to be correctly performed. Such operations included: adding a new row, adding a new column, changing the value of a cell, or changing the value of a formula. One error occurs when the participant does not perform one of those operations (e.g., the formula was not updated after inserting a new record), or the operation was performed incorrectly (e.g., the wrong value was introduced in the cell). The errors obtained correspond to the percentage of (sub)tasks that were not performed correctly.

In order to compare the results from each subject with the ones from the other subjects, only the subjects that performed all three tasks are show in the overall error rate charts shown in Fig. 13 and in Fig. 14. The results from the other subjects were excluded

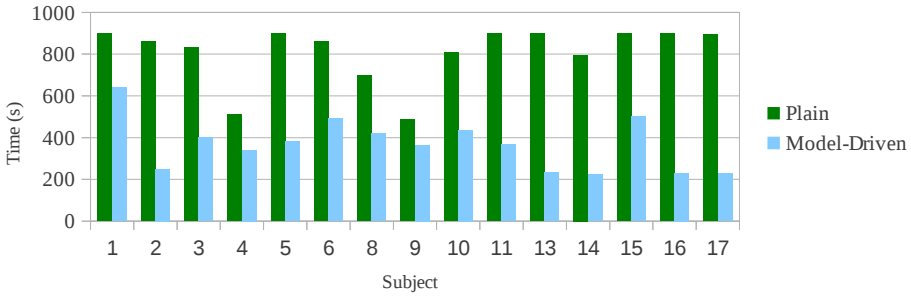


Figure 12: Time used to perform the tasks on the payments spreadsheet.

from this analysis since a given task may be more error-prone than another one, which could be misleading in the chart.

Error rates for the plain budget spreadsheet are around 50%, most of the errors being related to wrong formulas. Some errors are also present in the model-driven version of this spreadsheet, but they are in much lesser quantity since the environment deals automatically with the formulas. The errors present in the model-driven version (and also present in the plain one) result from the input of wrong values, or their input in the wrong places. This information is graphically shown in Fig. 13.

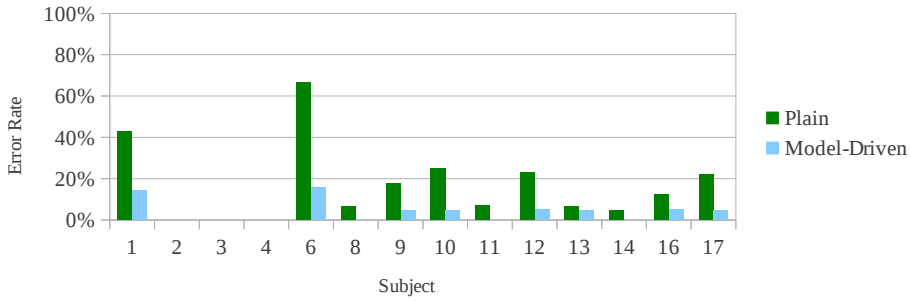


Figure 13: Error rate in the budget spreadsheet.

Similar results were obtained for the payments spreadsheet (see Fig. 14), with a slightly higher error rate for the plain version of the spreadsheet (around 60%).

This analysis was also performed at the task level, yielding similar results. They are not shown here since they do not bring any relevant information than what is already presented.

Subjects made mistakes in both plain and model-driven spreadsheets. Typical errors made in plain spreadsheets are the wrong or misplaced values, a wrong formula, and the lost of the spreadsheet structure. In the model-driven version the only type of errors are the wrong and misplaced values.

6.3.2 Hypothesis Testing

The significance level used throughout the evaluation of all the tests is 0.05. The evaluation of the tests was performed using the R environment for statistical computing [30].

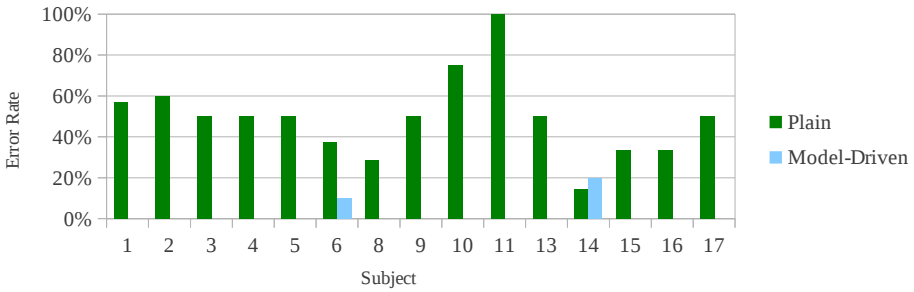


Figure 14: Error rate in the payments spreadsheet.

Comparison of times: The difference of times between the execution of tasks in plain spreadsheets and model-driven ones do not follow a normal distribution. Thus, we used the Wilcoxon test, which is the best fit for these cases [29].

The results obtained from the tests show that for model-driven spreadsheet development, in the particular case when our tool is used, the time taken to perform the tasks is statistically less than when using a plain spreadsheet (with p-value of 0.007882 for the budget spreadsheet and 0.0002441 for the payments one).

Comparison of error rates: The differences of error rates between plain spreadsheets and model-driven ones do not follow a normal distribution. Thus, we used a Wilcoxon test to test the null hypotheses for both spreadsheets, in order to be able to compare the results.

The results obtained from the tests show that for model-driven spreadsheet development, in the particular case when our tool is used, the number of errors are statistically less than when using a plain spreadsheet (with p-value of 0.0003579 for the budget spreadsheet and 0.001911 for the payments one).

6.4 Interpretation

The results from the analysis suggest that a model-driven approach to spreadsheet development can improve users' performance, while reducing the error rate. Moreover, from the questionnaires we can conclude that subjects felt more confident in the results of the model-driven approach compared to the the plain one. This indicates that some restrictions on the development process are welcome by the user since they understand this will make them perform better on their tasks.

6.4.1 Threats to validity

The goal of the study is to demonstrate a causal relationship between the use of a model-driven approach and improvements in the spreadsheet development process. Moreover, this study is defined to ensure that the actual setting used represents the theory we developed.

Next, validity threats for this study are analyzed, divided in four categories as defined in [31], namely: conclusion validity, internal validity, construct validity, and external validity.

Conclusion validity: The main concern is the low statistical power due to the low number of participants. To overcome this issue more powerful statistical tests were

performed where possible, taking always into account the necessary assumptions.

Problems related to measures can also arise, e.g., the times that the subjects took to perform the tasks. Nevertheless no significant differences to the real values are expected. Moreover, subjects performed the same tasks and in an environment that they are used to. Also, subjects have a similar background, which minimizes the risk of the variation being due to individual differences instead of the use of different treatments, but introduces problems when generalizing (see further on).

Internal validity: In order to minimize the effects on the independent variables that would reflect on the causality, several actions were taken. First, this study, with these subjects, is executed only once, some starting with plain spreadsheets and subsequently with the model-driven environment, while others starting with the model-driven environment and then working with the plain spreadsheets, with the goal to reduce learning effects. Second, the time to perform the study was reduced as much as possible so that the subjects could remain focused during all the study. Third, the instruments used (e.g., spreadsheets and questionnaires) were defined so that we could collect just what was needed. Fourth, all the subjects performed the same tasks, so issues from having different groups with distinct treatments do not arise. Specifying as much as possible the study, we obtained more control and reduced possible internal validity threats.

Construct validity: For this validity, several hypotheses to cover the aspects to analyze were defined with much detail, and mono-operation bias was reduced using two spreadsheets to collect data. Furthermore, the subjects were guaranteed to not be affected by this study, since they were not under evaluation (this was said to them several times during the study execution), and they were put at ease so that they would perform as much like in a real-world setting.

External validity: This validity is related to the ability to generalize the results of the experiment to industrial practice. For that, the spreadsheets used to collect the data were based on real-world ones. However, since this study was performed with a small homogeneous group, the results from this study cannot be generalized without analyzing the domain where to apply.

6.4.2 Inferences

Since this study was performed in a very specific setting, we cannot generalize to every case. Nevertheless, our setting was the most similar possible to a real one, where spreadsheets were based on real ones and Master students studies can come close to ones with professionals [32], so the results could be as if it was performed in a industry setting with professionals. This can bring the possibility that model-driven spreadsheet development can be useful, and studies in the industry can be used to assess the methodology in specific cases.

6.5 Discussion

The empirical study we conducted reveals very promising results for model-driven spreadsheets. Although participants had little time to learn model-driven spreadsheets, they completed their tasks faster and with less errors using such spreadsheets compared to the ones without models. This suggests that with little training users can greatly benefit from the use of models guiding them. Moreover, as we theorized, it was apparently easy to use the model-driven setting in an environment users are used to, that is, a spreadsheet system.

Nevertheless, from the study results we can also see how to improve our framework. The errors committed by participants in the model-driven environment were similar to the ones found in the plain spreadsheets. Two kinds of errors were found: values (correct or incorrect) inputted in the wrong cell and incorrect inputted values (in the correct cell). To improve the input values in the wrong place we believe we are in conditions of proposing some possible solutions:

- Labels' context: as often happens, the spreadsheets we gave to participants had blocks of cells repeated over columns/rows (e.g. payments over months, budgets over years). Given this scenario, it is quite easy to scroll over the spreadsheet and lose visual contact with labels, mainly in large spreadsheets. We believe that keeping labels always visible would help users to input the values in the correct cells. Thus, we plan to make labels as visible as possible. This can be achieve in two ways: first, one could use the spreadsheet mechanism that allows to keep some columns/rows visible always; second, given that our spreadsheets have a corresponding model where labels are known, MDSheet could repeat label cells every number of columns/rows it would see appropriate, or it could also suggest the person creating the model to repeat such labels when again it would see fit.
- Complete row/column context: we believe the previous solution can still be improved. Even with labels visible, inputing values "far" from labels can be error-prone. A possible helper mechanism would be to highlight the column/row corresponding to the cell the user is currently selecting. This would give extra context to the users' actions and would possibly minimize inputing values in the wrong cells.

Both solutions proposed need, of course, to be empirically validated. We plan to include them in MDSheet and pursue further validation both individually and in combinations.

To improve the second problem, the wrong values, we propose the integration of known techniques such as testing [33–39] and smell finders [40–43]. It is quite difficult to know if an inputted value is correct or not, but testing techniques could aid the user to know better. Moreover, smells can help users to search for places that can be potentially dangerous in the sense that errors can arise from those spreadsheet locations.

7 Related Work

In spite of its numerous benefits, model-driven engineering is sometimes difficult to realize in practice. In the context of spreadsheets, the use of model-driven software development requires that the developer is familiar both with the spreadsheet domain and with model-driven engineering. ViTSL [9] and *Gencel* [10] represent the first approach to deliver model-driven engineering to spreadsheet users. Using these tools, it is possible to create a model and to generate a new spreadsheet respecting it. This approach, however, has an important drawback: 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 automatic co-evolution of the model (instance) and its instance (model) is not possible. In our work we present a solution for these problems by embedding spreadsheet models under a spreadsheet system.

Hermans *et al.* [8] describe a technique to automatically infer class diagrams for existing spreadsheets matching them to a set of pre-defined patterns. The class diagram

inferred can then be used to further understand, improved, or re-implement the underlying spreadsheet. However, the relationship between the original spreadsheet and the inferred class diagram is then lost and no further connection exits between them. In our setting both model and spreadsheet are kept synchronized which allows the evolution of both artifacts.

Ko *et al.* [44] summarize and classify the research challenges of the end-user software engineering area. These include requirements gathering, design, specification, reuse, testing and debugging. However, besides the importance of Lehman's laws of software evolution [45], 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 formulas. Nevertheless, many of the transformations applied within spreadsheets originate in works aiming at spreadsheet generation.

Engels *et al.* propose a first attempt to solve the problem of spreadsheet evolution [46]. *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 technique to evolve spreadsheet models and instances in a different way: first, we use strategic programming [47] 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 [48] 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.

In this paper, we have studied the evolution of spreadsheet models and the coevolution of the corresponding instances. While the evolution of instances and coevolution of models has also already been realized at the theoretical level [49], the fact is that the proposed bidirectional engine is still under integration. So, for now, we focus on empirically evaluating the former setting, also for practical reasons: we believe that constructing an empirical validation scenario for the latter, more general, context would be unfeasible and potentially lead to undecipherable results.

The empirical study presented in this paper is partly based on the one from [50]. In their study, Carver *et al.* evaluate a methodology for spreadsheet testing and debugging, comparing it against the use of plain spreadsheets. Their study is mainly based on opinion gathering from the subjects, but they also evaluate two metrics: correctness and time. They conclude that the use of their methodology did not affect the correctness of spreadsheets created by users, but it did reduce the effort required to create them.

In fact, another study featuring spreadsheets based on models have been ran by the authors in the past [13, 14]. However such study considered a completely different setting of the one we now present. The models studied in the previous study are based on relational databases and not on *ClassSheets*. Thus, the spreadsheets created are not

comparable to the ones studied in this new work. Indeed the results from the previous study show that relational models for spreadsheets have several limitations while the new *ClassSheet* models have delivered much better results, as shown in Section 6. Moreover, the previous study does not consider the evolution of model-driven spreadsheets as it happens in the new study. Finally, the tools evaluated in the two studies are indeed different.

8 Conclusion

In this paper, we have presented techniques 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 embedding and co-evolution rules are available. Finally we assess the impact of this approach on users' productivity by performing an empirical study. The results obtained clearly show that spreadsheets under the model-driven setting are more reliable and faster to use than regular ones.

Given the promising results that we have observed, we plan to push model-driven spreadsheets further. Indeed, we now already have a complete bidirectional model-driven environment where users can evolve both the model and the data and having the corresponding artifact co-evolved [49]. In the future, we plan to empirically validate the usability and usefulness of evolving spreadsheet instances and having the corresponding model co-evolved. As we stated before these techniques must also be validated in an industrial environment. Thus, in collaboration with *Agere*, the municipal company that is responsible for supplying water to the city of Braga (which already supplied us one of the spreadsheets used in this study), we plan to run similar studies but now in a production environment.

Acknowledgments

The authors of this paper would like to express their gratitude to Dr. Nuno Alpoim, CEO of *Agere*, for providing us and our study with a spreadsheet under usage in industry.

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project `FCOMP-01-0124-FEDER-010048`. This work was also supported by Fundação para a Ciência e a Tecnologia with grants `SFRH/BPD/73358/2010` and `SFRH/BPD/46987/2008`.

References

[1] F. Hermans, M. Pinzger, and A. van Deursen, "Supporting professional spreadsheet users by generating leveled dataflow diagrams," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 451–460.

[2] B. A. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*, 1st ed. Cambridge, MA, USA: MIT Press, 1993.

[3] R. R. Panko and N. Ordway, "Sarbanes-oxley: What about all the spreadsheets?" *CoRR*, vol. abs/0804.0797, 2008.

[4] R. Panko, "Spreadsheet errors: What we know. what we think we can do." *EuSpRIG*, 2000.

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

[6] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert, "Visual specifications of correct spreadsheets," in *VL/HCC*. IEEE Computer Society, 2005, pp. 189–196.

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

[8] 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.

[9] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger, "Automatic generation and maintenance of correct spreadsheets," in *ICSE*. ACM, 2005, pp. 136–145.

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

[11] J. Cunha, J. Mendes, J. P. Fernandes, and J. Saraiva, "Embedding and evolution of spreadsheet models in spreadsheet systems," in *VL/HCC '11*. IEEE, 2011, pp. 179–186.

[12] J. Cunha, J. Visser, T. Alves, and J. Saraiva, "Type-safe evolution of spreadsheets," in *FASE*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 186–201.

[13] L. Beckwith, J. Cunha, J. P. Fernandes, and J. Saraiva, "End-users productivity in model-based spreadsheets: An empirical study," in *IS-EUD*, 2011, pp. 282–288.

[14] ——, "An empirical study on end-users productivity using model-based spreadsheets," in *Proceedings of the European Spreadsheet Risks Interest Group*, ser. EuSpRIG '11, S. Thorne and G. Croll, Eds., July 2011, pp. 87–100.

[15] P. Stevens, J. Whittle, and G. Booch, Eds., *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2863. Springer, 2003.

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

[17] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, "Extension and implementation of classsheet models," in *Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VLHCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 19–22.

[18] J. Cunha, J. P. Fernandes, and J. Saraiva, "From Relational ClassSheets to UML+OCL," in *the Software Engineering Track at the 27th Annual ACM Symposium On Applied Computing (SAC 2012)*, Riva del Garda (Trento), Italy. ACM, March 2012, pp. 1151–1158.

[19] S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, Eds., *Advanced Functional Programming, Third International School, Braga, Portugal, September 12-19, 1998, Revised Lectures*, ser. Lecture Notes in Computer Science, vol. 1608. Springer, 1999.

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

[21] J. Oliveira, "A reification calculus for model-oriented software specification," *Formal Asp. Comput.*, vol. 2, no. 1, pp. 1–23, 1990.

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

[23] A. Cunha, J. Oliveira, and J. Visser, "Type-safe two-level data transformation," in *Proc. Formal Methods, 14th Int. Symp. Formal Methods Europe*, ser. LNCS, J. Misra *et al.*, Eds., vol. 4085. Springer, 2006, pp. 284–299.

[24] A. Cunha and J. Visser, "Strongly typed rewriting for coupled software transformation," *ENTCS*, vol. 174, no. 1, pp. 17–34, 2007, 7th Workshop on Rule-Based Programming.

[25] ——, "Transformation of structure-shy programs: applied to XPath queries and strategic functions," in *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, G. Ramalingam and E. Visser, Eds. ACM, 2007, pp. 11–20.

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

[27] S. Peyton Jones, G. Washburn, and S. Weirich, "Wobbly types: type inference for generalised algebraic data types," Univ. of Pennsylvania, Tech. Rep. MS-CIS-05-26, Jul. 2004.

[28] R. Hinze, A. Löh, and B. Oliveira, ""Scrap your boilerplate" reloaded," in *Proc. 8th Int. Symposium on Functional and Logic Programming*, 2006, to appear.

[29] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, ser. Computer Science. Springer, 2012.

[30] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org

[31] T. D. Cook and D. T. Campbell, *Quasi-experimentation: design & analysis issues for field settings*. Houghton Mifflin, 1979.

[32] M. Höst, B. Regnell, and C. Wohlin, "Using students as subjects – a comparative study of students and professionals in lead-time impact assessment," *Empirical Software Engineering*, vol. 5, no. 3, pp. 201–214, Nov. 2000.

[33] G. Rothermel, M. Burnett, L. Li, and A. Sheretov, "A methodology for testing spreadsheets," *ACM Transactions on Software Engineering and Methodology*, vol. 10, pp. 110–147, 2001.

[34] M. Fisher II, M. Cao, G. Rothermel, C. Cook, and M. Burnett, "Automated test case generation for spreadsheets," in *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*. New York: ACM Press, May 19–25 2002, pp. 141–154.

[35] R. Abraham and M. Erwig, "Autotest: A tool for automatic test case generation in spreadsheets," in *VL/HCC*. IEEE Computer Society, 2006, pp. 43–50.

[36] M. Fisher II, G. Rothermel, D. Brown, M. Cao, C. Cook, and M. Burnett, "Integrating automated test generation into the WYSIWYT spreadsheet testing methdology," *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 2, pp. 150–194, April 2006.

[37] R. Abraham and M. Erwig, "Goaldebug: A spreadsheet debugger for end users," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 251–260.

[38] ——, "UCheck: A spreadsheet type checker for end users." *J. Vis. Lang. Comput.*, vol. 18, no. 1, pp. 71–95, 2007.

[39] ——, "Mutation operators for spreadsheets," *IEEE Trans. Software Eng*, vol. 35, no. 1, pp. 94–108, 2009.

[40] J. Cunha, J. P. Fernandes, J. Mendes, H. Ribeiro, and J. Saraiva, "Towards a Catalog of Spreadsheet Smells," in *The 12th International Conference on Computational Science and Its Applications*, ser. ICCSA'12, vol. 7336. LNCS, 2012, pp. 202–216.

[41] F. Hermans, M. Pinzger, and A. v. Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 441–451.

[42] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *ICSM*, 2012, to appear.

[43] A. Asavametha, "Detecting bad smells in spreadsheets," Master's thesis, Oregon State University, 2012.

[44] 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.

[45] 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.

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

[47] R. Lämmel and J. Visser, "Typed Combinators for Generic Traversal," in *Proc. Practical Aspects of Declarative Programming PADL 2002*, ser. LNCS, vol. 2257. Springer, Jan. 2002, pp. 137–154.

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

[49] J. Cunha, J. P. Fernandes, J. Mendes, H. Pacheco, and J. Saraiva, "Bidirectional Transformation of Model-Driven Spreadsheets," in *Theory and Practice of Model Transformations – ICMT 2012*, ser. Lecture Notes in Computer Science, Z. Hu and J. de Lara, Eds. Springer-Verlag, 2012, vol. 7307, pp. 105–120.

[50] J. Carver, M. Fisher, II, and G. Rothermel, "An empirical evaluation of a testing and debugging methodology for excel," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ser. ISESE '06. New York, NY, USA: ACM, 2006, pp. 278–287.